PARC ENVIRONMENT ON SUN NETWORK

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

BY

Milind A. Bhandarkar

to the
Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
May, 1991.

CERTIFICATE

It is certified that the work contained in this thesis entitled *ParC Environment on Sun Network*, by *Milind A. Bhandarkar* has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

(Dr. R.K.Ghosh)
Asst. Professor
Dept. of Computer Science
and Engg.
I.I.T. Kanpur

(Dr.P.Gupta)
Asst. Professor
Dept. of Computer Science
and Engg.
I.I.T. Kanpur

CSE-1991-M-BHA-PARC

To

rsesun2 l

Who maintained its identity

in spite of

being in the network.

Acknowledgements

I wish to express my sincere thanks to my supervisors Dr. P. Gupta and Dr. R. K. Ghosh who introduced me to the fascinating field of parallel processing and provided me with this opportunity to work in this field. Dr. S. Saxena helped me from time to time and was always willing to listen to me during this work.

Prasad Rao, T. Srikanth, Ravi Shankar, Borah & Sriram provided me with the right atmosphere to work in. Atul Tulshibagwale taught me many a "trivial" thing. Antony, Bhatta, Srinivas, Narayana, Shreesh and other C.S. guys were quite enthusiastic while discussing anything academic or non-academic.

I thank the Hall-IV ghat 'aish'ociation, especially Makya Joshi, Wadya, Tatya, Vinya, Kasrya & Baman for the great time I had in I.I.T.K. Pankha gave me the images to work on. nrg@eevax2 provided pleasant diversions from the work. I am thankful to Suvarna for some great debates we had on various issues.

I know that there are many friends without whose direct or indirect involvement, this work would not be a reality. I express my gratitude towards all of them.

ABSTRACT

In this thesis, the implementation and application of ParC language on the Sun workstation network at Dept. of CSE, I.I.T., Kanpur is discussed. ParC is a distributed programming language which is available on transputer networks and which can be used for parallel computations utilizing the computing power of idle workstations. The ParC implementation is based on the concept of a communication server. It employs a pseudo-busy-waiting strategy for message-passing and avoids the synchronization problems of inter-task communication. Programs written in this language on Sun network are easily portable to any transputer-based system because of the source-code compatibility. A general purpose image processing system is built on this parallel platform as an illustration of the capabilities of the platform and is reported in this thesis. This system uses the master-slave paradigm for performing image processing operations in parallel.

CONTENTS

Chapter 1

Introduction

Speed is one of the most important criteria for determining the performance of a computer. As the technology is progressing, faster and faster computers are becoming available commercially. But it seems that the limit to the speed of computaion will be reached soon and still, some problems will remain to be solved because of the time constraints. Therefore in the past decade, the developments in computer architecture have shifted their focus from sequential to parallel processing. Parallel processing exploits inherent independence in operations that are done in some order on traditional sequential computers and uses more than one processing units to do these operations concurrently. Computers have been developed which utilize multiple processing elements. These can be classified into groups according to various parameters in different ways. Some of the popular classification schemes are due to Flynn, Feng and Handler. [Hwang 85]

1.1 Motivation

With the advent of fast and cheap microcomputers and communication hardware, the networks of diskless workstations are fast becoming popular as an alternative to costly parallel architectures such as vector computers like Cray. In the department of CSE at IIT, Kanpur, we have such a network of 12 Sun workstations. Since there was no image processing software available on this network, our aim is to provide a library of image processing routines. But most of the tasks in image processing being compute-intensive and availability of the processing power of all the 12 workstations motivated us to develop a parallel platform using the low level network programming facilities on which such an image processing system could be developed. Since transputer-based systems hope to provide an elegant solution for the development of parallel programming environments [Jesshope89] [Walker85] [Transputer89a] [Transputer89b], we wanted our platform to be closely similar to the transputer-based message passing architectures so that any applications developed on this platform be ported to transputers without much difficulty.

1.2 Distributed Languages

The dividing line between distributed and parallel languages is not fixed and there are no clear distinguishing factors between them. According to most, the difference lies in the tightness of coupling of processors. But distributed languages like Occam [Inmos84] have been existent on both tightly and loosely coupled multiprocessors. However, the purpose of this section is not to provide the reader with a comparison of the distributed and the so-called parallel languages but to discuss the features of distributed languages.

Distributed languages differ from the traditional sequential languages in (i) use of multiple processors, (ii) co-operation among the processors and (iii) potential of recovery from partial failures [Bal89]. The first factor prompts for parallelism in applications of distributed languages. Parallelism is expressed in various ways in distributed languages. Most natural unit of parallelism is processes which are completely sequential programs running on a uniprocessor and which have their own data and stack. The processes may be created dynamically at the run-time but in some languages the number of processes in an application are determined at the compile time. According to [Bal89] this makes mapping processes to physical processors easy. Also the distributed languages make use of the operating system primitives to deal with process creation and termination. Some other units of expressing parallelism are objects in PRESTO [Bershod88] and SINA [Tripathi89], parallel statements in OCCAM [Inmos84], functional parallelism and AND/OR parallelism [Bal89]. In some languages where a distributed application is expressed as a single process, threads or recursive agents are used for specifying parallelism [Hansen89]. But such languages are best suited for implementation on shared memory multiprocessors. ParC [Inmos89] uses explicit process declaration and also supports the use of threads within a process to express concurrency.

The second issue has to do with communication. Communication is defined as exchange of data between different logical processors in [Dubois88]. According to [Bal89] the co-operation between two processes

involves communication and synchronization. According to [Dubois88], synchronization is a special form of communication in which the data are the control information. Several communication and synchronization mechanisms are available in different programming languages. The common ones are synchronous and asynchronous point to point messages, rendezvous, remote procedure calls, broadcast messages, shared variables, semaphores, mutexes, condition variables, global names such as tuple space [Bal89].

The third issue deals with recovery from partial failures. Several approaches are taken by designers of various distributed languages on this issue. Some of them are (i) to ignore failures altogether (ii) detection of failure by run-time system or the operating system (iii) repetition of primitive such as a remote procedure call (iv) replication of units such as processes to restart the failed task. For having fault tolerance in the application written using a distributed language, it is very important that all the transactions are atomic.

We now brief the reader with some examples of distributed languages. One of the first successful attempt at designing a distributed language that can be implemented on a variety of architectures was CSP [Hoare78]. This prompted the design of many CSP-like languages. Some examples are Occam [Inmos84], CSP-i [Wrench88], Joyce [Hansen87b] [Hansen89a] [Hansen89b], ParC [Inmos89]. Although CSP has been highly successful as a notation for theoretical work, it is far too removed from the requirements of a secure programming language, according to [Hansen87a]. A program written using CSP-like language uses a fixed number of processes which can be determined at compile-time. The communication between them takes place through channels and is syncronous. That is, the process executing a send or a receive is blocked until the process at the other end of the channel executes the complementary statement. Guarded statements which allow nondeterminism to be expressed in the program are allowed in Occam and Joyce but not in ParC. There have been attempts to generalize Occam to use bidirectional guards which are reported in [Bornat 86]. Attempts to extend the CSP concepts to include control over

scheduling of processes were also made considering nondeterminacy as a positive disadvantage [Kerridge86]. Occam and ParC were originally developed for transputer-based systems but Occam, in particular, has been implemented on variety of architectures [Cooper88][Fisher86]. LOTOS is a specification language for distributed programs which has been influenced by CSP and is based on Calculus of Communicating Systems [Logrippo88]. In LOTOS the inter-processor communication occurs by 2-way rendezvous called interaction through Occam-like channels which are called gates. Languages which are based on the rendezvous concept of Ada form another class of distributed languages. Some of the examples of this class are BNR Pascal [Gammage87], Concurrent C [Gehani86] and Concurrent C++ [Gehani88]. Edison is a distributed programming language which has been implemented on the network of microcomputers based on Zilog Z-80 [Dubnicki88]. An exhaustive survey of all distributed languages is available in [Bal89].

1.3 Image Processing Systems

Traditionally, image processing for various applications such as remote sensing, medical imaging, robotics was done on sequential computers. As the applications increased and the necessity of real-time processing was felt, the tasks involved in image processing being compute-intensive, special hardware for image processing was built and the architecture of these special systems utilised the 'natural' parallelism involved in the operations. On sequential machines, many of the applications were developed using libraries for image processing routines like SPIDER. SPIDER [Tamura83] is a subroutine package for image data enhancement and recognition. This has about 400 different routines for performing various tasks. This was developed for use with FORTRAN language because most of the work in image processing was being done in that language. Before this also some libraries like VICAR (Jet Propulsion Laboratory) and SLIP (Nagoya University) existed but since the routines in those libraries were not free of I/O, it hampered their portability. Alongwith SPIDER, an interactive package, CUPID (Conversation mode Utilities for Processing Image Data) was built which made use of most of the SPIDER routines.

The main reason for a general purpose image processing system on

any parallel architecture not being popular was the non-portability of applications and the 'software inertia', since most of the applications were written to be suited to sequential machines [Reeves84]. Also one of the reasons was that different image processing applications require different types of parallel architectures so that they become most efficient. For example the SIMD, wavefront and systolic array processors are suited for low level vision and MIMD architectures are suitable for higher levels like recognition. The non-availability of an architecture-independent programming language became the main stumbling block in the way of a general purpose image processing system on parallel computers [Reeves84]. An attempt was made on a Unix based uniprocessor system to treat all the processes involved in image processing as filters, thus giving them the uniformity that was essential [Landy84]. Also, based on abstract specification languages for image processing applications, expert systems were built which used the paradigm of knowledge-based composition of image analysis programs [Matsuyama89]. Systems which combine image processing with data-bases of images were built where the images were classified into a number of types such as intensity images, convolution images, regions, contours and slope frames. A hierarchical data-base of images was also built [Vernon88]. Even recent studies show that the use of parallel computers for image processing is limited to some specific applications and special architectures are devoted to these systems [Hall89][Hamey89]. For low level vision, an architecture independent language called 'Apply' was developed and tested for comparison with existing libraries like SPIDER. The results are reported in [Hamey89][Wallace89]. This language is implemented on a number of different parallel architectures like Warp(CMU) and other Unix-based uniprocessors. This work claims that adapting an algorithm to a particular parallel architecture poses a significant barrier to the vision programmer, and hampers the portability of the application. The language 'Apply' provides the facility to specify operations for some window which can be done in an architecture-independent manner. This operation is then performed on all such windows of an image. Thus high-level vision algorithms cannot be coded in 'Apply'. Transputer-based systems go one step further by

providing the language Occam. Programs written in Occam preserve their semantics when run on networks with different numbers of transputers. This can help significantly in developing architecture-independent general purpose image processing systems. PIPS and ImagePro are the examples of two such systems [Srivastava90] [Udupikar91]. Another approach to architecture-independent parallel computation which can be used for image processing is given in [Skillicorn90].

1.4 Thesis Organisation

The thesis is organised into 4 chapters. The next chapter gives the concepts behind ParC and its implementation on the Sun Network. Chapter 3 describes the design of Imagetool, a parallel image processing system. We conclude in chapter 4 after briefing the reader about future work. Manuals of ParC run-time support, Configuration language, lightweight processes and Imagetool are attached as appendices.

Chapter 2

ParC Platform : Implementation

2.1 Introduction

After its launch in October 1985 by INMOS Ltd, UK, the transputer was described to be the most significant event in concurrent computing, perhaps comparable to the advent of microprocessors in sequential computing. The Transputer was designed to be used as a basic element in high performance parallel systems. It is a single chip component which has a fast processor (10 MIPS), 2K/4K of on chip memory and significantly faster communication links (with a speed of 20Mbits per second). Each transputer has 4 links which provide for connections in multi-transputer networks. Each link consists of two unidirectional channels which perform synchronized communication. Thus a transputer can be used as a stand-alone processor or as a processor in the network communicating with other processors. The transputer also offers the facility of running a number of concurrent processes. For these processes, the communication takes place through channels which are basically words in its memory. An intraprocessor channel can be identified by the word address. There is no difference between the interprocessor and intraprocessor channels from the programming point of view. Therefore it is natural to have the same functional interface for communication between processes without knowing where they are placed. This has a great impact on the process of software development with transputers. The semantics of programs written for transputers remain same regardless of the number of transputers in the system. This improves the portability of applications written in ParC or OCCAM.

The transputer has a built-in scheduler which increases the speed of context switching etc. For example, Unix scheduler takes about 100 microseconds to switch from one process to another which the transputer does in about 3 microseconds. Each process on the transputer can have one of two priorities, *urgent* or *not-urgent*. A not-urgent process may get descheduled because of the end of its timeslice, whereas an urgent process runs uninterrupted till it relinquishes the control voluntarily or blocks

for communication.

The additional facilities available on the recent transputers are a 64 bit floating point unit, a digital signal processing unit, high performance graphics support and internal timers.

Design of the transputer and therefore ParC, is heavily influenced by the concepts of Communicating Sequential Processes.

In sequential programming languages, a program is viewed as a sequence of instructions or statements which, when executed in that order, gives the desired output. The idea of Communicating Sequential Processes (CSP) captures the sequencing of instructions in sequential programs, adding to it the concept of communication between them. This idea appears very natural because of its occurrence in everyday business in the real world. A large task is split among many working units (human beings for example) which, while continuing to do their work independently, communicate and decide the course of further actions depending on the contents of the communication.

In the CSP paradigm, a computing system is assumed to be a group of sequential processes communicating with each other using communication channels. Each channel connects exactly one process to another process. Each channel carries information in only one direction and if bidirectional communication is required between any two processes, it must be achieved using two channels. Each process can have a number of input and output channels. This number is fixed in advance and can not be changed during the course of execution. Communication across channels is synchronized, i.e. a process sending data over a channel has to wait until the data is received at the other end.

This can be illustrated by a collection of n processes numbered from 0 to n-1. The 0^{th} process generates input for the next process and writes it onto its output channel. The $(n-1)^{th}$ process consumes the output of its previous process, read from its input channel, by, say, writing to the disk. In the computation, there are n-2 stages, each of which is performed by processes numbered from 1 to n-2. Each process reads the input from

its input channel and after performing computation, writes the output on the output channel. The system is configured in such a way that, except for the last process, the output channel of i^{th} process is connected to the input channel of $(i+1)^{th}$ process.

This example illustrates one more encouraging fact about this paradigm that a process is viewed here as a black-box and the internal behaviour of any process is not important to the rest of the components of the system. This facilitates in building systems which are recomposable [Tulshibagwale90].

ParC is based on the abstract model of CSP. Each computing system is called an application, which consists of one or more tasks. Each task is similar to a UNIX process, having its own code, data and stack segment. It has a number of input channels, also called input ports and a number of output channels, also called output ports. In Inmos ParC, these are passed as arguments to the main function, similar to the command line arguments argc and argv. But in our implementation, they are maintained as global variables with the same names i.e. "ins" and "outs". "in_ports" and "out_ports" are global variables denoting the number of input and output ports respectively.

The first statement of any task in ParC in our implementation should be a call to START() which initializes some variables and establishes connections between processes. This is not a requirement in the Transputer ParC.

Besides the intertask channels, which can not be changed, created or destroyed during the execution of the task, there can be a number of intratask channels which can be created at run-time. These channels are uninitialized and should be initialized before their use. Any channel is referred to with a pointer to the channel word (CHAN *). The tasks which are placed on different transputers can be connected only when the physical processors are connected with wires. When a number of processes on a single transputer want to communicate with processes on other transputer, they have to share the physical connection. Since the

transputer does not provide facilities for sharing the links meaningfully and without collisions, one must write one's own multiplexer task for sharing the links.

The number of tasks on a particular transputer is limited by the memory available. There can be a number of different execution threads within a task. Threads are subprocesses which have their own stack and therefore independent local variables which are created on the stack. This stack is allocated at the time of creation of a thread. All the threads in a single task share the data area and the code of the task to which they belong. Thus all the threads running within a particular task run on the same processor on which the task is placed. Similar to the tasks, the threads also have a priority, which can be *urgent* or *not-urgent*. The scheduling of threads is similar to the task-scheduling described earlier. Threads can communicate among themselves using channels or shared memory. Semaphores are provided for synchronization of access to shared data or shared channels.

As is common to all concurrent programming languages, the incorrect use of powerful mechanisms of threads and semaphores may lead to problems which are not detectable at the compile time. Therefore a certain discipline is required in use of these features. Note that a deadlock can occur because of synchronized communication if it is not used properly and that there exist no special deadlock detection mechanisms in the transputer version of ParC. But, on the Sun Network, this implementation supports the deadlock detection.

Programs written in ParC should be compiled into executable code before using them as components of an application. For a single-transputer system, this is sufficient to run the application. But for a multi-transputer system where, to achieve maximum performance, one needs to distribute the tasks over a number of processors, it is required to configure the application according to the underlying network.

The configuration language recognized by the general purpose configurer "config" is designed for the above purpose. It takes as input a

file containing the following, coded in the syntax of "config" language:

* transputers in the physical network
* physical connections between transputers
* definitions of tasks
* the connections between tasks
* placement of tasks
* bindings of channels

The config program loads the executable files in the transputers of the network and thus sets up the application. For a complete description of the configuration language syntax on the Sun network, please refer to the appendix on config.

ParC has been implemented on the Sun network. The following section describes the implementation in detail.

2.2 Implementation of ParC on Sun Network

The implementation of ParC platform on Sun network consists of the development of a run-time library for ParC which provides the functional interface to the parallel features of ParC; a remote execution server and an interpreter for the configuration language. The following subsection briefly describes the networking facilities provided by the underlying Sun network. Later in this section, we give details of the implementation.

2.2.1. Sun Network at IIT,Kanpur

The Department of Computer Science at IIT, Kanpur has a network of 12 Sun 3/50 3/60C workstations connected by 10 Mbits-per-second Ethernet. There are three file servers and a network transparent file system. Since the processor (68020) and the operating system (SunOS 4.0.3) are the same on every node, it represents a homogeneous distributed computing environment.

As there is no utility to write distributed programs using the computational facilities of the whole network, the necessity was felt to

provide one and the ParC platform which is described in this report offers the solution to this problem. One more reason for developing ParC platform on this network was the low CPU utilization. As is common with most computing facilities, a computationally cheaper task such as editing files etc. which does not take much CPU time continues for most of the time [Shoja88], whereas some applications like image processing, when run on a single processor, take enormous time. Thus, it was found necessary to distribute the computationally expensive tasks over the network so as to increase the CPU utilization.

SunOS provides low-level primitives such as socket-based interprocess communication and remote procedure calls to achieve distribution of a set of tasks in an application over the network. It also provides mechanisms to get the distribution of load on the nodes in the network. But it slows down the program development because it leaves everything to the programmer. Thus, if a programmer wants to write a simple utility to add, say 1 million numbers stored in a file in a distributed manner, he has to take care of distributing tasks, establishing connections, communicating using sockets, combining results and the synchronization using the low-level primitives. A platform like ParC provides an easy way to do this leaving most of the tasks to the system.

The ParC platform is developed using the remote procedure call mechanism of SunOS [Sun88]. The RPC (remote procedure calls) mechanism is a high-level communication paradigm. It assumes the existence of some low-level mechanism like TCP/IP or UDP/IP for socket-based communication. This makes program development easy for applications which employ the client-server model of communication. The server is a process which offers and co-ordinates access to some resource, whereas clients are processes which want to access these resources.

When a client makes a remote procedure call identifying the server and the service it wants to avail, it sends a data packet to the server which is waiting to receive it. The server then decodes the data packet to identify the service to be performed, performs the service and sends the data packet containing the results back to the client. The client is

suspended until the result is received. The server and the client can be (and usually are) on two different machines. On a heterogeneous network, since the internal data representations might be different for client and server, before passing the parameters of the remote procedure call to the client through the socket, they are converted to a unique representation called XDR (external data representation). On the server, it is again converted into its internal data representation. The results sent by the server are also converted to XDR which are decoded again at the client site. Thus RPC appears to be the same as a normal procedure call except for the fact that it is executed in the address space of a different process which may be on a remote machine. RPC mechanism is described graphically in fig 2.1.

The ParC platform on this network also assumes that the file system structure on every workstation is same. This is achieved by Sun's Network File System (NFS). This is a facility to share files even in a heterogeneous environment of machines, operating systems and networks. It also provides methods for crash recovery and transparent access with a high performance due to its integration with the SunOS kernel.

2.2.2. Threads

Threads in ParC on Sun network are implemented using the Sun lightweight processes (lwp) library described in the appendix C. Most of the functions required by the ParC threads interface have equivalents in the lwp library. But because of the different scheduling policy of Sun lwp's it was required to modify them to suit our requirement. Also a number of threads are required for other management purposes. For example, a high priority scheduler thread which performs the basic task of reshuffling the queue of threads in the not-urgent category is created in *START()*. This thread sleeps for a certain predecided time and on waking up, if a not-urgent thread is running, puts it at the end of the queue allowing other threads in the same category to continue execution. In this way we achieve the scheduling policy used by transputers. The other example of such *system* threads is the thread which controls communication with timeout. When a thread requests communication with timeout, one such thread is

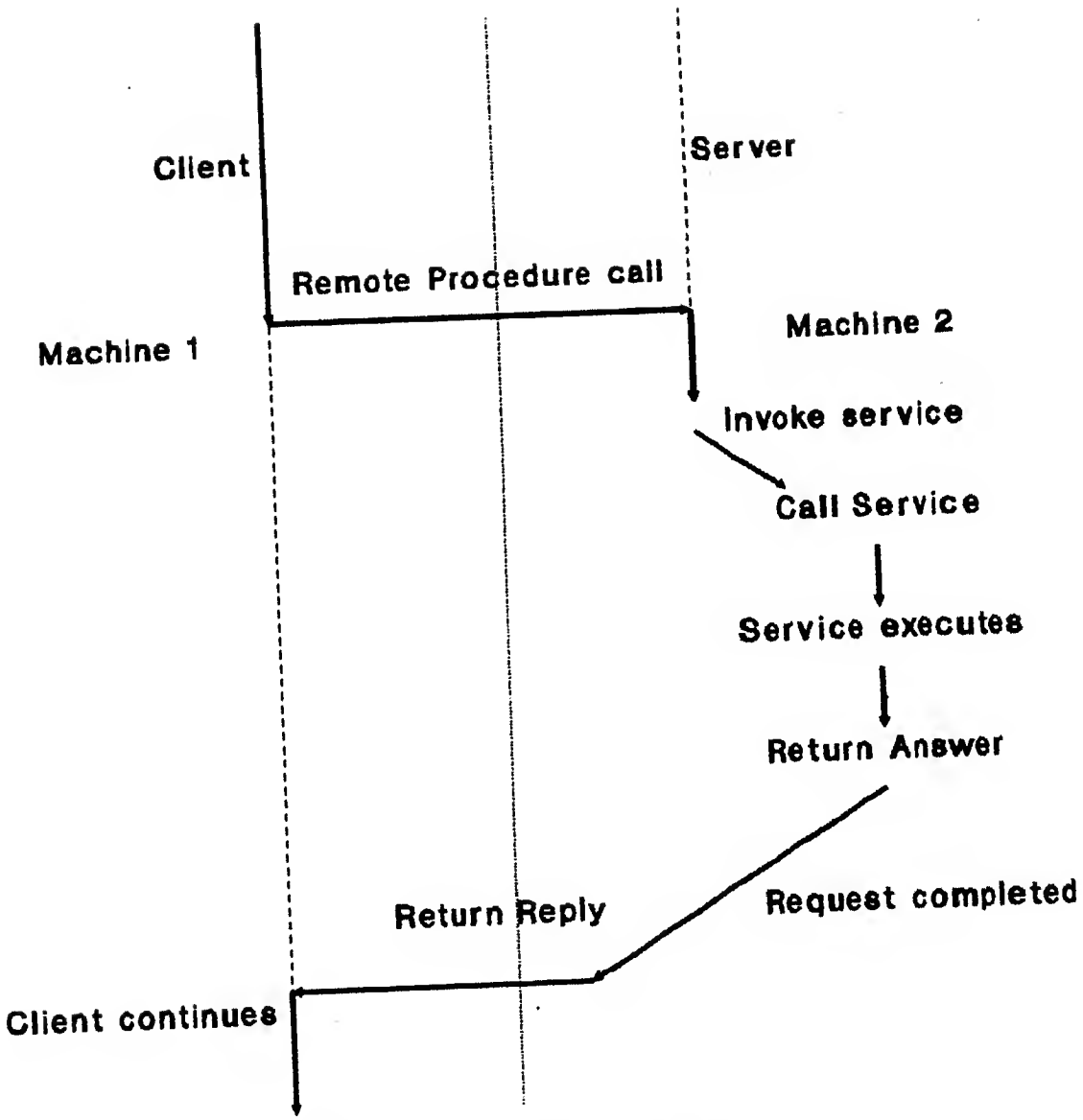


Fig. 2.1 Sun RPC Mechanism

created which sleeps for the timeout specified and after waking up checks whether the communication is already achieved. If not, it wakes up the sleeping thread with an error condition, otherwise goes to sleep.

2.2.3. Communication

The communication in ParC is achieved through unidirectional channels which are used to pass messages of arbitrary lengths with synchronization. The channels are implemented as structures (of the type *CHAN*) which contain a unique identifier, space for the message to be sent or received, length of message and a lock for synchronization.

This is different from the transputer implementation of channels where they are represented just as a word in main memory, because the transputer provides built-in mechanism for synchronization and also a way to write into or read from the address space of another process which is not possible in the UNIX environment. Since the functional interface to access channels is the same irrespective of whether they connect two tasks on the same machine or different machines or two threads in the same task, the *CHAN* structure also contains an indication of their being intratask or intertask. If the channels are local to a task, i.e. connecting two threads in the same task, then the *CHAN* structure also contains space for the thread identifier which is waiting for communication across that channel. As we differentiate between the intratask and the intertask communication at the implementation level, we describe them below separately.

Intratask communication refers to communication across channels that connect two threads belonging to the same task. Since these two threads have common data area, the message is copied just once which makes this communication faster.

When a thread requests communication, a function *comm* is called which checks that the channel is an intratask one and after verifying that, it calls a function *intracomm*. This function, *intracomm*, checks the type of request which can be input or output. If it is an output request, it checks if any other thread is waiting for input on that channel. If no such thread

exists, then this function puts the requesting thread in a suspended state. But before that, it puts the identifier of that thread into the channel on which output is to be done. If some thread is already occupying that channel for output, this function returns with an error because a channel is supposed to be shared by only two execution entities. For one, it is an input channel whereas for the other it is used for output. If some thread is waiting for input, this function copies the message into the area specified in the channel, wakes up the thread waiting for input and returns success to both of the threads.

If the request for communication is for input, the function *intracomm* checks for any thread waiting for output. If some thread is waiting for input over the specified channel, it returns with an error because input channel-sharing by two threads is a violation of the basic model of CSP. In transputer ParC this may cause the system to crash. If no thread is waiting for output on that channel, this function copies data address and the number of bytes to be transferred into the channel structure along with the thread identifier of the requesting thread. It then puts this thread in a suspended state so that it would be woken up once the data is available on the channel. If some thread is already waiting for the data to be delivered at the time of this request, *intracomm* copies the specified number of bytes into the data area specified along with the input request, and wakes up the thread from which this data has arrived.

Note that when any operation is requested on a channel it is first locked. When the thread requesting communication goes into suspended state, this channel is released. Thus, there is no possibility of deadlocks because of context switching while a thread is in the critical region for accessing channels. When the number of bytes requested and sent does not match, the behaviour of transputer ParC is unspecified whereas in our implementation, it treats this as an error condition.

The other type of communication is Intertask communication which refers to communication between two threads which belong to different processes (tasks). The case of intertask communication is very much different than the intratask communication. Since the two tasks

communicating over a certain channel do not have access to each other's data space, this cannot be done with the algorithm described previously. Several approaches were considered for implementing this. Some of them are described below.

One of the simplest solutions was to implement intertask channels using files. The filename of the particular channel will be known to tasks at both its ends. Also a particular record will be written and read by the tasks which contains the message and its length. But this approach fails in synchronizing the two tasks in an efficient way. Also there is no way to authenticate that the records written in the file belong to only that process which belongs to the application. In the UNIX kernel, because of the buffer cache used, the data written on a file does not immediately get to the disk, hence there will be one more overhead, that of flushing the buffer each time after writing to the file. There will also be problems of deadlock detection because of the distributed nature of communication.

Another solution was to use sockets to implement intertask channels. Though this would eliminate the problems of authentication, the other problems still remain. Again there would be no efficient ways to synchronize and to detect deadlocks. As in the previous case, this has a limitation on the number of open files permitted by the Unix OS. In a processor farming application, where a single manager distributes the work packets to all the tasks, meaning that it should be connected to all the tasks, this would mean a number of open sockets for the manager task which would not be permissible.

Thus we want a mechanism by which the communication would be efficient, simple deadlock detection would be possible and should not be affected by UNIX limitations, whatever the size of the application may be. We now describe such an approach.

To overcome the limitations described above, the idea of a communication server is proposed. Each application consisting of a number of tasks will have one communication server for synchronizing the communications over intertask channels. This reduces the number of file

descriptors open for a particular task because irrespective of the number of input and output channels, each task has just one pair of sockets connecting itself to the communication server (or *commserver* in short). The commserver is implemented using the remote procedure call mechanism of SunOS.

Each time a task wants to communicate over a channel it will make a remote procedure call to the communication server, the handle of which is passed to it through the environment. Before making any RPCs it first establishes connection with the *commserver*. This is done in the *START()* routine transparently. Thus a task does not need to know the existence of a *commserver* at ParC level. The *commserver* which is waiting for any RPC to arrive, recognizes the channel over which the task wants to communicate from the data packed as parameters to this RPC and checks if any other task has requested communication over that channel. If it is so, then it sends a message in the case of a receive and copies the message down in the case of a send from the parameters. It returns with the indication of success. If no other task has requested communication, then it sets the flags in the internal channel structure indicating that some task has requested communication, and sends a reply (*ENOTYET*) to the task which is waiting for communication. Then the local communication routine, which is *intercomm* in this case, relinquishes the control of the thread in the task which has requested communication and again, when it is scheduled, sends a renewed request.

At this point, it is necessary to explain a few things about the basic algorithm of the *intercomm* routine. It employs a *pseudo-busy-waiting* strategy for synchronization. That is, if no task is waiting on the other side of the channel, it deschedules the thread requesting communication allowing other thread in the same category to continue. This strategy seems to be the most efficient in multi-threaded programs. The other approach is to suspend the thread requesting communication until the task on the other side of the channel requests communication, and *commserver* sends a signal to this task indicating the same. Though this approach, at first sight, looks more efficient, has several implementation problems. The

most important of them is the possibility of deadlock if proper care is not taken in implementation.

Consider the following sequence of actions: A thread in task 1 requests communication over an intertask channel. Since there is no task waiting on the other end of the channel, *commserver* sends *ENOTYET*. The local routine, *intercomm*, decides that the thread is to be put in the suspended state. The task at the other end, task 2, requests communication. The *commserver* sends a signal *SIGURG* to task 1 indicating that a request has arrived. The *SIGURG* handler, unaware that the thread has not yet been suspended, tries to wake it up. The *intercomm* routine in task 1 which now gets the control, suspends that thread. This will lead to a deadlock situation. Though this can be avoided by tough concurrency-control measures such as introducing an intermittently executing thread checking for consistency, it will be very inefficient. Therefore our policy of just descheduling the thread proves to be quite efficient. For writing efficient programs in ParC, it is suggested that a number of threads should be simultaneously existent so that the other threads can continue computing while a thread tries to communicate.

In the implementation of *commserver*, tables are maintained for both input and output channels. There is a table for each task which indicates the number of input and output channels and also their connections. Each task is identified with a unique id called the *clientid* supplied to it by the configurer. A channel in the *commserver* can be in any of the following states:

* Clear i.e. Unoccupied (*CHANCLR*)
* Send requested (*SNDREQ*)
* Receive requested (*RCVREQ*)
* Send satisfied (*SNDSENT*)
* Receive satisfied (*RCVRECD*)

The transition graph of the channel states is shown in fig. 2.2. Here *S* denotes a send request and *R* denotes a receive request.

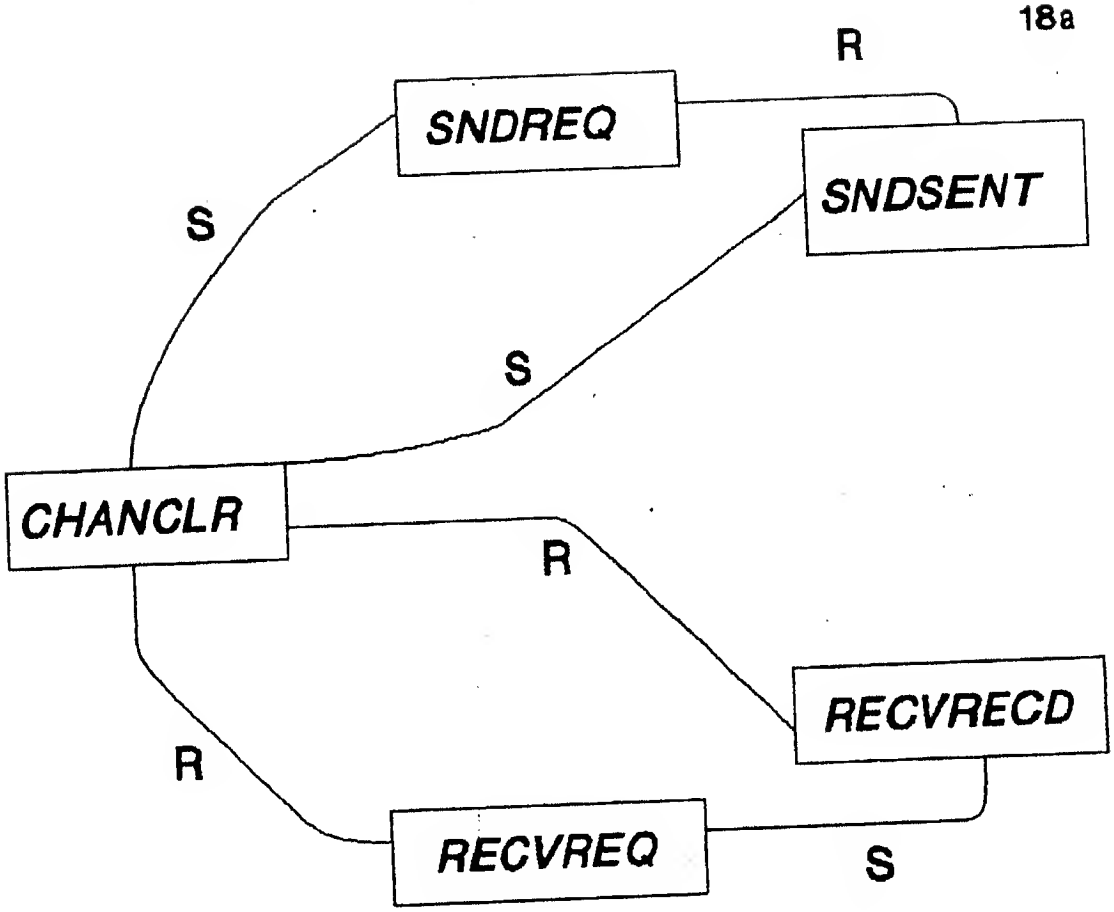


Fig. 2.2 Transition State Diagram for Channels

The *commserver* also supports other remote calls like cancel communication, reset channel, connect channels and create channels. The latter two are used by the *config* program.

2.2.4. Remote Execution

In multi-transputer applications, different tasks often need to be put on different processors. Thus from the program running on the host machine, it should be possible to remotely execute a program on other nodes in the network. Unlike transputers, where the program is downloaded from the host's memory to the transputer, here we rely on the network file system which is accessible to all the nodes on the network. Thus the executable image of the program to be run on the remote machine should be accessible on that machine. Also in this implementation it is required that the parameters such as the number of input ports, the number of output ports, the name of the machine on which the commserver is running and the *URGENT* attribute in the configuration language (see Appendix B) should be passed to the remote program so that they can be retrieved by the *START()* routine to set up the connections etc.

Sun network services provide for such a server and can be communicated to by the *rexec* call. But there are limitations of this. For example one can not pass the environment to the remote process. This can be overcome by executing a special program with command line arguments containing the name of the program to be executed and the environment to be set up. The remote execution server offered by SunOS forks a shell on the remote machine which in turn forks a process. Thus the number of processes running on a remote machine is double the number of tasks, since there is one extra shell per process.

This is avoided by implementing our own remote execution server. This server, like the *commserver*, is implemented using RPC. The environment to be set up for the remote process is passed as arguments in remote procedure calls. The *rex* server sets up the environment and forks another process. It uses a *vfork* system call instead of *fork* which is very efficient because it does not involve copying of data space. Instead, when

a child process is *vforked*, it works in the area of the parent process till it calls *exec* or *exit*. Till then, the parent process does not get control. As we set up the environment before forking, the only statement that the child executes is *exec*, hence the parent data space is not corrupted.

2.2.5. config : A configurtion language

Finally, the interpreter for the configuration language *config* is implemented. This reads a file of the form *.cf which contains the program written in config. (See Appendix B) It also reads a file called *sites.db* to read the site definitions. Then it establishes connections with all the sites on the network. If some sites are not working due to some reason, *config* removes their definitions from the internal database.

After checking the application for syntax, it maps the logical processors specified in the *.cf file to actual processors. This mapping is done taking into account the load on each of the sites. Thus the processor is mapped to the site on which there is the least load. In this implementation, *config* considers all the sites to be having the same computational power but in future this is likely to be changed by including a performance coefficient in the sites.db file for all the physical nodes. After mapping, it forks the *commserver* and creates channels. It also issues the connect requests to *commserver* giving the channel numbers and task numbers to identify channels. Finally, it executes all the tasks remotely and waits for all of them to finish. If any error condition occurs in any of the task, it is immediately received from the *rex* server and then requests to kill the tasks at all the sites are issued to the *rex* servers after giving appropriate messages to the user.

A number of good features like that of process migration are likely to be implemented in *config*. It will provide for the load balancing on the network. If a certain site is heavily loaded, we may be able to monitor that and migrate the tasks running on that site to other available sites. Also, in this implementation there are no means provided for crash recovery which can be provided in the future versions.

Chapter 3

ImageTool : An Application of ParC

3.1 Introduction

Digital image processing is concerned with processing a two-dimensional picture by digital computer [Gonzalez77] [Rosenfeld76]. With the advent of sophisticated optical devices, it has found applications in such diverse areas as Robotics, Remote sensing, Medical sciences, Food technology and Experimental stress analysis to name a few. A digital image is a two-dimensional array of integers. Each of the element of the array is called a picture element, pixel or pel. Each pixel has certain value which indicates the luminance and called as the gray value. Typical image data consists of 512 X 512 picture elements. In many of the applications, the image obtained from a satellite or any other optical device (like a CCD camera) is to be processed in real-time giving results immediately. An example of such an application is "Obstacle detection" in Robotics. A Robot is equipped with a camera which captures and sends the image of Robot's path to the processor. This image is to be processed and analysed for occurrence of any obstacle so that further decisions regarding the movement of the Robot can be taken immediately.

Considering the size of the input to the processor, the traditional SISD architectures do not serve the purpose. Also, since most of the image processing operations exhibit 'natural' parallelism because of locality of decisions etc., it was inevitable to use parallel architectures for image processing. Some of the most common architectures used for real-time image processing are *wavefront array processors*, *systolic processors* and other SIMD architectures. But in other applications where the processing is not required to be real-time, the special-purpose processors for image processing like the ones mentioned before are not found cost-effective and hence the MIMD paradigm is used. Imagetool, a general purpose image processing system, which this chapter describes, is built using the above model.

All the image processing operations can be divided into 4 types [Dougherty87].

1. Image representation

2. Image-to-Image transformation

3. Image-to-parameter transformation and

4. Parameter-to-decision transformation.

The first category of operations includes conversion of two-dimensional optical data to its digital counterpart. This involves sampling, selection of (x,y) coordinates to specify the intensity levels and quantization, selection of discrete values for representing continuous intensity values so that error is minimized.

However, most of the image processing operations come under the second category, the image-to-image transformations. Image restoration, the process of deblurring the image; Image enhancement, emphasizing certain features in an image to convert the image to a form better suited to human or machine analysis; and Image segmentation, separating the objects of interest from the rest of the picture constitute most of the operations in this category.

Once the enhancements to the image are done, one can extract or compute the features from the enhanced image. These features are computed as numerical values and hence the third category, image-to-parameter transformations. This also causes data compression since these operations extract only those data which are required for further decisions.

The fourth and the last category includes the operations where, based on the features extracted, decisions about the inclusion of the objects of interest are taken. This is referred to as classification. In all the applications of Image processing, this operation is an important one and is used for different objectives. For example, in Remote sensing, this may be used to determine the type of forestation or in Defence-related applications it might be used to detect the presence of air strips etc.

Imagetool deals only with the last 3 types of transformations. It provides many operations of the second category and also can be extended

to include operations of the third and fourth categories. Since the image-to-image transformations are fairly standard for all the application areas, we have included only these operations in Imagetool, leaving the scope for application-specific extensions.

Following section describes the architecture of Imagetool in general. Later in this chapter, various functions of Imagetool are discussed.

3.2 Imagetool : System Overview

Imagetool is a general purpose image processing system based on the ParC platform described in the earlier chapter. This section describes the design philosophy and gives overview of the system.

Imagetool consists of three components. These are listed below:

1. Itool , a user interface

2. Iserv , a server for image processing operations

3. A number of subservers.

These three components form the three layers of Imagetool. Fig(3.1) shows the interconnections of these layers. We describe them from the top layer to the bottom.

3.2.1 Itool : A user interface

Itool is based on the popular windowing package Sunview running on SunOS. It offers a user-friendly interface to the image processing library functions offerred by the other two components. This layer assumes the existence of the other two layers and does not use any of the features of ParC. Thus, this package can be run on any machine of the network which can connect to the machine running Iserv by sockets. This tool is written using the language C. The layout of the screen of Itool is shown in fig. 3.2.

The user interface can be divided into 5 parts. These are,

1.Input File Specifier

2.Output File Specifier

3.Display area

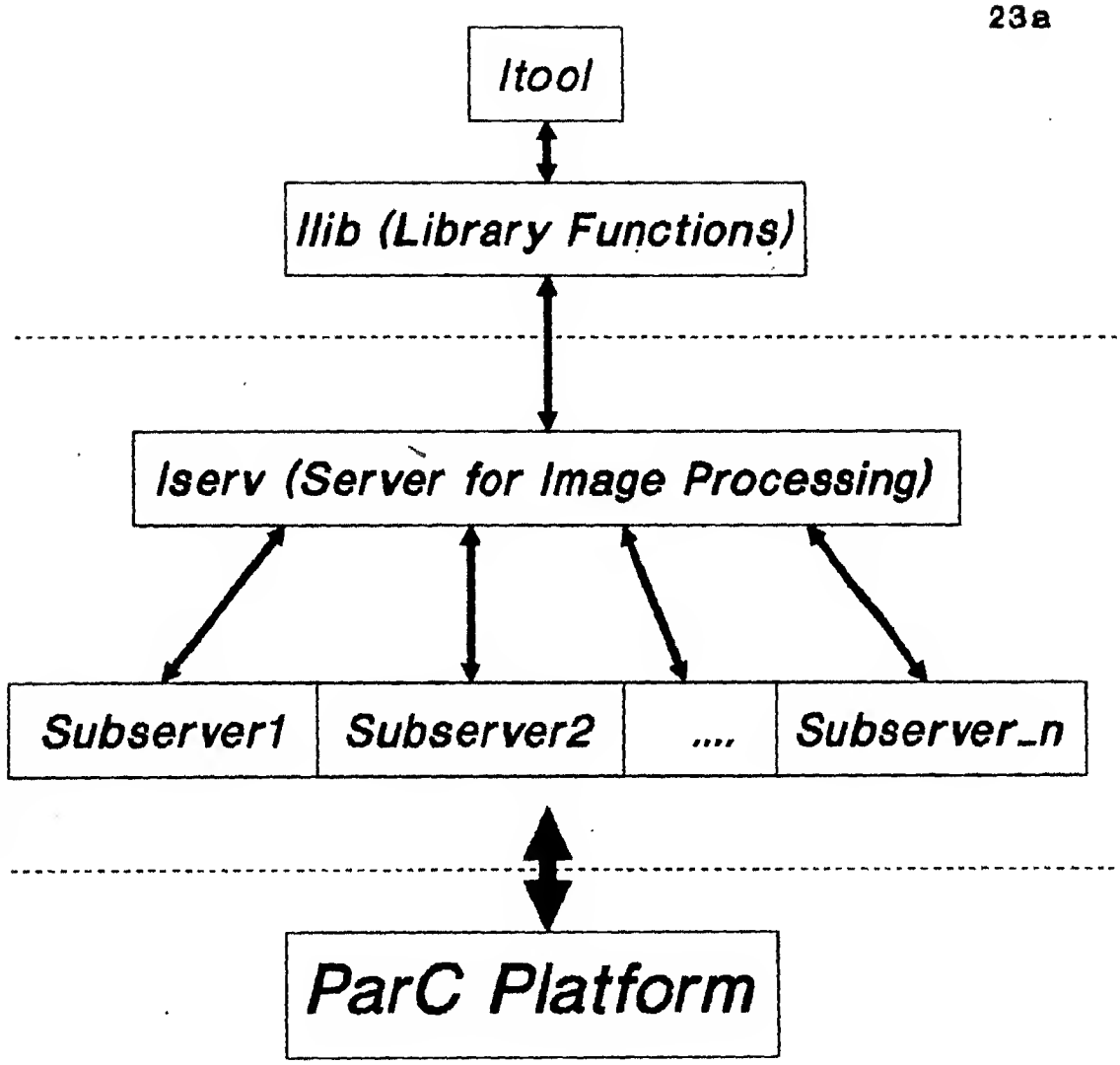


Fig. 3.1 Structure of Imagetool

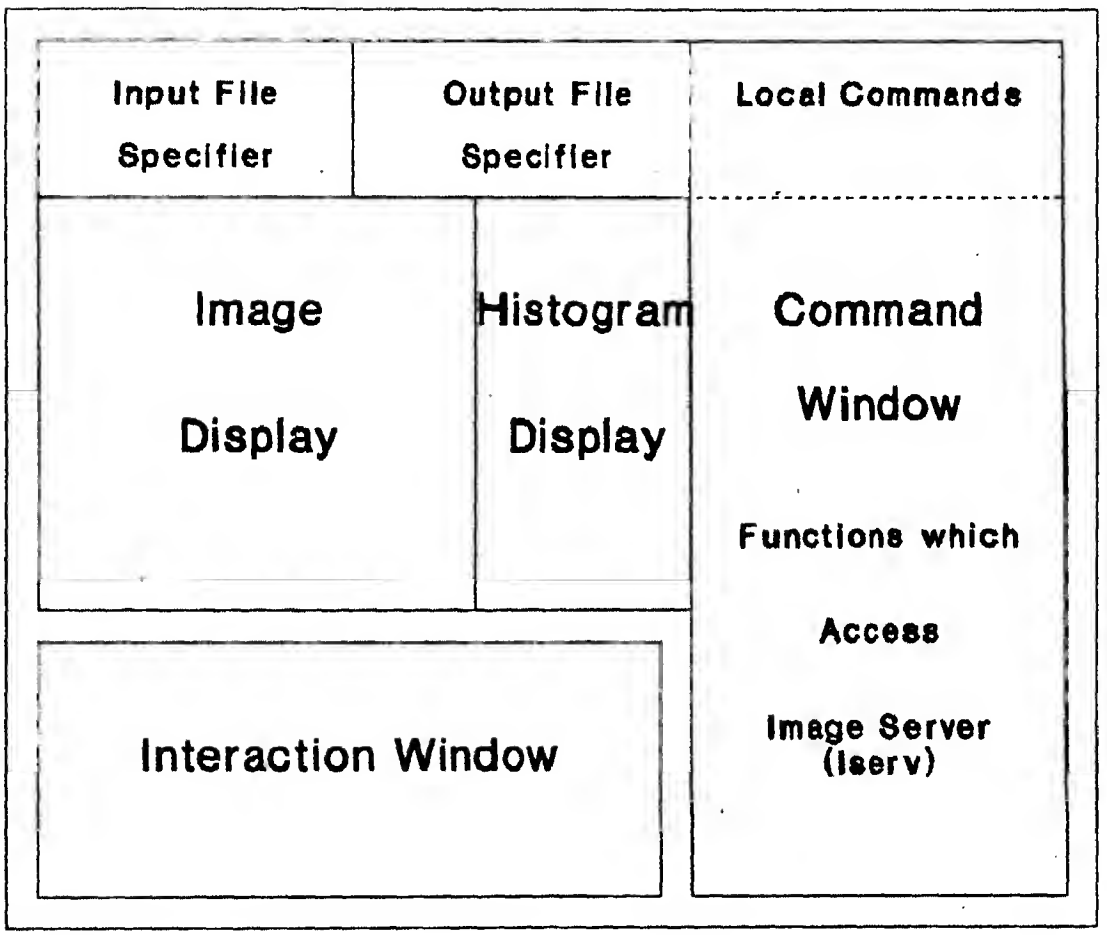


Fig. 3.2 Layout of Itool

4.Command window

5.Window for messages.

In Itool, it is assumed that the image is stored in a file on the file system which is common to all the three layers of Imagetool. The image is stored as a sequence of characters with no separators. One byte is allocated for one pixel which implies that a pixel can have at most 256 gray levels. This is common to most image acquisition systems and small utilities can be written to convert images with different format to this format. No information regarding the number of scanlines (rows of image) or pixels per scanline is stored in the image file so as to confirm with majority of imaging systems. However, routines can be provided to access a small image database which stores this information.

Thus any image can be identified with its complete Unix filename, Number of scanlines, Pixels per scanline and number of gray values. This constitutes a file specifier in Imagetool. There are two file specifier windows in Itool. They are Input File Specifier and Output File Specifier. As their names suggest, the Input File Specifier information is used in all the *LOAD* operations, whereas the Output File Specifier information is the basis for all *STORE* operations. Since an *XCHG* (Exchange) operation includes LOAD as its part, the Input File Specifier is used in that operation also. If a certain file can not be accessed because of either the nonexistence or the read-prohibition; or the specified amount of data cannot be retrieved from that file, appropriate error message is displayed at the top of that file specifier.

The display window is used to get the visual representation of the image. This window is further divided into two windows: the image window and the histogram window. Maximum image size permitted to be loaded into the dislplay buffer as well as for any other operation is 1024 X 1024. But only one fourth of that area i.e. 512 X 512 is visible at any time to the user because of the limitations of hardware. To overcome this, two scrollbars are provided at the top and left of the window so that the image can be scrolled and the user can see the entire image part-by-part. Beside the image display window is the histogram window. A histogram is a

frequency diagram where, on Y-axis, frequencies of the gray values are plotted against the gray values themselves on the X-axis. The maximum frequency is printed at the top of the window. In the background of the histogram the X-Y plane is seen divided so as to enable the user to calculate the frequency of a particular gray value as compared to the maximum frequency.

The fourth and most important component of the Itool is the command window. This window gives the user access to the image processing routines offerred by the image server. These operations can be accessed by taking the mouse to the button with required label in command window and pressing the left mouse button. Since the routines executed by the image servers in parallel will be discussed in the next section, here we describe the facilities available to the user locally , i.e without the use of Iserv.

The local functions are listed at the top of the window. These are various screen control functions (clear, redraw), display operations dealing with image files (load, store, exchange), help for the beginner and program invocation and termination (shell, edit, quit). Please note that the Load, Store and Xchg do their respective tasks in local space and have no impact on Iserv and other subservers. Load operation reads the Input file specifier and fills the display buffer with the data read. Also it finds the histogram of the read file and displays it in the adjoining window. The Store operation similarly reads the Output file specifier and stores the image in the display buffer into that file. It does not clear the display buffer, whereas the Xchg operation exchanges the input and output file specifiers and performs a Load operation.

3.2.2 Iserv : An Image Processing Server

Iserv, which provides user programs with access to the image processing services available, acts both as a manager for the subservices and as a gateway to the client programs. It is designed in ParC in such a way that extensions are easily possible. This is very important as there is an enormous number of image processing operations which are used in

practice. It is difficult to provide all these in a single library. Also, providing just a small number of basic, commonly used operations causes the system to be too restricted. Providing the facility to combine these basic operations to achieve application specific tasks is a good solution but selection of the set of operations which provides a complete basis for all operations poses a problem [Walters87]. Thus we chose to keep this server, Iserv, configurable while offering the services through user-level functions so that a number of basic functions can be combined to perform an application-specific task which is not provided by Iserv.

Iserv, being a manager for all the subservices, is connected to various instances of subservers running at different sites with communication channels of ParC. The ParC application consisting of Iserv and other subservers which is specified in the Configuration language is run by the interpreter for configuration language, config. Iserv must be placed on the machine from which the user invokes the application. This is done by specifying that Iserv must be placed on 'host' in configuration language.

When Iserv starts execution, it first tries to communicate over all the communication channels that it has and receives the information about the subservices offered by different programs connected at the other ends of these channels. There may be a number of subservers offering the same subservice. The information it receives from any subserver consists of (i) Number of subservices offered (ii) Channel number to be used for further communication (iii) Number of inputs, outputs and their formats for each subservice (iv) An indication of whether the image is to be divided for that subservice. This information is called a *protocol* and since different subservices require different inputs and provide different outputs, Iserv has to manage multiple protocols. It then forms a lookup table of these protocols and refers to it when it receives a request from any client to perform a certain subservice.

Another important task of the Iserv program is to divide the image and distribute different parts of it to the different subservers so that they can be processed independently. For example, if there are 4 instances

of, say, Histogram modification server which offers the service, Normalization, which can be done independently on different parts, Iserv sends a record consisting of start and end positions of the part , so that the operation is performed on only that part by an instance of the Histogram Modification Server. Image division is done according to the number of subservers available, so that ultimately the whole image is processed. Iserv recognizes all the basic data types of C and also provides a number of other often required data structures like file specifiers and masks. Also, new data structures can be introduced without altering the original code significantly.

When a number of subservers are available and the whole task is divided into subtasks to be performed in parallel, these multiple servers output different data which are to be combined in some way and sent back to the client as a result of the entire operation. Iserv performs this task. It has a small routine to combine the individual results for each data structure. For example, most of the operations return 0/1 depending on the failure/success of the operation. This is given a special data type in Iserv. This data type is combined in such a way that the result is a multiplication of all individual results. This serves our purpose because of the assumption that the entire operation is successful only when all the individual operations are successful. Similarly, the data structure for histogram is also combined so that the result is the histogram of the entire picture.

Iserv communicates with the client program, like Itool, using a standard Unix Socket. For this purpose we use the AF_UNIX stream socket of name "img_socket" in the directory from where Itool is run. Thus the user has to make sure that Itool and Iserv must be executed from the same directory. This socket is used by Itool to specify the subservice to perform, to communicate the inputs required by the subservers and to receive the outputs of these operations. Each subservice is specified by a 2-tuple, ⟨serv,subserv⟩. Where serv is is the number denoting the server offering the subservice subserv. 'serv' is a positive number and -1 is used for requesting Iserv to terminate a session.

However, the clients do not specifically have to read and write from sockets because of the image processing library provided to perform subservices (see Appendix D). For example, the client can use the image averaging subservice by calling the function image_avg() with appropriate parameters. The image_avg function would write the server and subservice number and the parameters to the socket and wait for reading results from the socket and then return those results back to the calling program.

3.2.3 Subservices

As mentioned before, the architecture of Imagetool is easily extensible. The subservices form the lowermost layer of the Imagetool. The subservices are the image processing operations grouped according to some general structures of routines called the services. In Imagetool, the subservices offered are grouped into 3 categories. They are (i) Convolution and related operations, (ii) Point based operations and (iii) Histogram modification. In this section, a general structure of the subservers is explained and later all the operations available in Imagetool are briefly discussed.

Any subserver is a ParC program which offers a number of subservices and has an input and an output channel to connect itself with the manager Iserv. One or more instances of this subserver can be connected to Iserv. After the initialization required for any ParC program which is performed in the startup routine START(), this program outputs the protocol of various subservices on its output channel. The protocol consists of the parameters that are required to perform an image processing operation and the outputs that these routines produce. It also outputs the indices of subservices along with the protocol for identification. After this, the subserver waits to read a subservice number from its input channel. When a legal number is received, it reads the appropriate parameters from the input channel, calls the corresponding routine to perform operation and writes the output on the output channel. The operation is performed in the local space of this program. If the subservice number received is (-1), this program exits. Since the subservice numbers are recieved from Iserv, this program exits only when

the Iserv sends (-1) as subservice numbers on its output channel. This is done when Iserv receives a 'close' call from Itool.

Another common feature of all the subservers implemented is the concept of buffers as storage space for images. Though this is not an essential condition for all the subservers, they are implemented because of the ease of composing a larger application from primitive functions with minimal intermediate storage. Each subserver maintains a number of internal buffers for storing images or their parts. Thus each server offers three basic operations to Load a buffer from an input file, Store a buffer into a file, and Copy one buffer to another. All the image processing operations now take integer parameters indicating buffer numbers and operate on these buffers only. Each buffer includes a header containing the number of scanlines, pixels per scanline and number of gray levels of the original image. It also contains the specification of the part of image allocated to it, i.e. the starting scanline, pixel and number of scanlines, pixels.

(I) Convolution & Related Operations

Two-dimensional convolution of functions g(x,y) and f(x,y) is defined as

$$g(x,y)*f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(\alpha,\beta)f(x-\alpha,y-\beta)\, d\alpha d\beta$$

The discrete counterpart of two-dimensional convolution is

$$g(x,y)*f(x,y) = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} g(m,n)f(x-m,y-n)$$

This is a very important operation in image processing and is used in most of the image-to-image transformations, specifically in image restoration, enhancement and segmentation. In the above equation we have the function f as the image we want to operate on and the function g varies for different applications. Usually function g is defined over a very small area and it is convenient to think of that area as a neighbourhood of the pixel which has some effect on the value of pixel after this operation is performed. Function g is usually referred to as a mask. The convolution server uses

3X3 predefined masks for all operations but a general routine which lets the user specify the mask is provided. It can be used with the name *image_con3()*. The convolution or masking can be illustrated by the following equation. If the mask, i.e. the function g, is:

$$\begin{bmatrix} w_0 & w_1 & w_2 \\ w_3 & w_4 & w_5 \\ w_6 & w_7 & w_8 \end{bmatrix}$$

Then the pixel value f(x,y) is replaced by

$$f(x-1,y-1)*w_0 + f(x-1,y)*w_1 + f(x-1,y+1)*w_2 + f(x,y-1)*w_3 + f(x,y)*w_4 + f(x,y+1)*w_5 + f(x+1,y-1)*w_6 + f(x+1,y)*w_7 + f(x+1,y+1)*w_8$$

The convolution server in Imagetool offers a number of operations based on convolution. These are described below.

(I.a) Smoothing

Noise may be present in an image because of either a poor sampling system or a faulty transmission channel. The effects of this noise are lessened if not removed using the smoothing operators. The masks used for this purpose are called spatial averaging masks because the operation is done in the spatial domain. Spatial averaging is a process of low-pass filtering which causes some distortion blur in the image because of some high-frequency components. Imagetool offers three functions for smoothing using convolution. They use following masks:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

These functions are called *image_avg()*, *image_nc1()*, *image_nc2()*.

(I.b) Median Filtering

Though median filtering is not a convolution-based operation it is offered by this subserver because of the structural similarities with other convolution-based routines. This method is used for smoothing when the noise in the image consists of intensity spikes and crispened edges are to be preserved. Median filtering consists of replacing the pixel value by the median of its neighbourhood values. For this we sort the neighbourhood for each pixel incrementally while moving from one pixel to another and replace the pixel value by the midpoint of the sorted sequence. In imagetool it can be availed through function *image_median()*.

(I.c) Edge Crispening

A natural photograph or visual signal is psychophysically more pleasing to the eyes if it has accentuated edges. Therefore edge crispening is used in the printing industry to sharpen the edges . For this purpose, a suitable gradient based on its neighbourhood is added to each pixel value which acts as a high-pass filter. Usually, a Laplacian gradient is added. Imagetool provides three different sharpening routines. These are *image_sh1()*, *image_sh2()*, *image_sh3()*. These routines use the following masks in that order.

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & -2 & 1 \\ -2 & 5 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

(I.d) Edge Detection

An edge is a boundary or contour indicated by a significant change in some physical aspect of the image or image attribute like the luminance level or the texture. Edge detection is a very important preliminary operation in every image understanding and image segmentation system. In Imagetool, we consider an edge to be a boundary between two regions of distinct gray level properties; that is, only the edges formed because of luminance difference, called the luminance edges are considered. The process of luminance edge detection is based on computation of a local gradient which indicates the rate of change of luminance along a line i.e. based on directional derivatives. Another way to find edges is to find out

the Laplacian operator which is defined below:

$$L[f(x,y)] = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

For digital pictures the gradients are approximated by first differences Δ_x , Δ_y or Δ_θ which are defined as :

$$\Delta_x f(x,y) = f(x,y) - f(x-1,y)$$

$$\Delta_y f(x,y) = f(x,y) - f(x,y-1)$$

$$\Delta_\theta f(x,y) = \Delta_x f(x,y) \cos\theta + \Delta_y f(x,y) \sin\theta$$

The second order differences are:

$$\Delta^2_x f(i,j) = \Delta_x f(i+1,j) - \Delta_x f(i,j) = f(i+1,j) + f(i-1,j) - 2f(i,j)$$

$$\Delta^2_y f(i,j) = \Delta_y f(i+1,j) - \Delta_y f(i,j) = f(i,j+1) + f(i,j-1) - 2f(i,j)$$

Therefore, the Laplacian operator is approximated as

$$\nabla^2 f = \Delta^2_x + \Delta^2_y$$

That is,

$$\nabla^2 f(i,j) = f(i+1,j) + f(i-1,j) + f(i,j+1) + f(i,j-1) - 4f(i,j)$$

Compass gradient masks are used to detect edges where the luminance changes in a particular direction. Imagetool offers all the compass gradient masks in addition to three Laplacian masks and four directional gradient masks. The routines which are available and the masks which are used in Imagetool for edge detection are listed below.

Directional gradients:

The routines for directional edge detection supported by Imagetool are *image_horfil()*, *image_verfil()*, *image_diafil1()*, *image_diafil2()*. The masks they use are:

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix} \begin{bmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{bmatrix}$$

Laplacian Edge detection:

There are three routines available for Laplacian edge detection depending upon the different approximations of Laplacian operator. These are *image_lp1()*, *image_lp2()*, *image_lp3()*. They use the following masks:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

Compass gradient masks:

Imagetool supports all the eight compass gradient masks. They are available through routines *image_cg_n()*, *image_cg_e()*, *image_cg_w()*, *image_cg_s()*, *image_cg_ne()*, *image_cg_nw()*, *image_cg_se()*, *image_cg_sw()*. The masks are:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & 1 \\ -1 & -1 & -1 \end{bmatrix} \begin{bmatrix} -1 & 1 & 1 \\ -1 & -2 & 1 \\ -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & -1 \\ 1 & -2 & -1 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} -1 & -1 & -1 \\ 1 & -2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 \\ -1 & -2 & 1 \\ -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & -1 \\ 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} -1 & -1 & 1 \\ -1 & -2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 & -1 \\ 1 & -2 & -1 \\ 1 & 1 & 1 \end{bmatrix}$$

(I.e) Nonlinear edge detection methods

Like the median filtering, the nonlinear methods for edge detection are also not based on convolution, but since they are also local neighbourhood operations, the structure of the routines is similar to the convolution-based operators. Hence they are clubbed together with the above operations. Imagetool offers two nonlinear edge detection methods which are popularly called the Sobel and Prewitt operators. These are also gradient-based approaches to edge detection but differ in the way they estimate the gradient. Here ,

$$f(i,j) = [\, X^2(i,j) + Y^2(i,j)]^{1/2}$$

$$X(i,j) = X_1(i,j) - X_2(i,j)$$

$$Y(i,j) = Y_1(i,j) - Y_2(i,j)$$

In Sobel operator,

$$X_1(i,j) = f(i-1,j+1) + 2f(i,j+1) + f(i+1,j+1)$$

$$X_2(i,j) = f(i-1,j-1) + 2f(i,j-1) + f(i+1,j-1)$$

$$Y_1(i,j) = f(i-1,j-1) + 2f(i-1,j) + f(i-1,j+1)$$

$$Y_2(i,j) = f(i+1,j-1) + 2f(i+1,j) + f(i+1,j+1)$$

Whereas, in Prewitt operator,

$$X_1(i,j) = f(i-1,j+1) + f(i,j+1) + f(i+1,j+1)$$

$$X_2(i,j) = f(i-1,j-1) + f(i,j-1) + f(i+1,j-1)$$

$$Y_1(i,j) = f(i-1,j-1) + f(i-1,j) + f(i-1,j+1)$$

$$Y_2(i,j) = f(i+1,j-1) + f(i+1,j) + f(i+1,j+1)$$

These operators are available in Imagetool as routines *image_sobel()* and *image_prewitt()*.

(II) Histogram Modification

A histogram, or the frequency diagram of gray values is a description of the luminance pattern in the whole image. Usually in images received from sources such as satellites, the histogram is highly skewed to the lower end, which is darker. Therefore, majority of pixels have below-average brightness levels. For better visualization of images, the contrast needs to be enhanced so that the details hidden in the darker side of the brightness range become visible. Linear contrast streching does not prove adequate because it spaces all the gray levels equally, thus making no change in the image if some pixels in the original image already have the extreme gray values. For such images, the histogram modification techniques are used.

The histogram modification is done in such a way that, given a particular target histogram, original image is modified so as to make its histogram as close as possible to the target histogram subject to the following constraints:

1. All the pixels with a gray value x are mapped to the same gray

value y everywhere in the image.

2. If gray value x_1 is mapped to y_1 and gray value x_2 is mapped to y_2 and $x_1 > x_2$, then $y_1 \geq y_2$.

3. Also the new gray values should be in the range 0-(L-1) where L is the maximum number of gray values in the input image.

Histogram modification to a given target histogram is achieved using histogram equalization as a primitive. Histogram equalization is a special form of histogram modification where the target histogram is uniform. If the source histogram is denoted by $p_r(r_k)$, n_k is the frequency of r_k and n is the total number of pixels, the gray value r_k is mapped to s_k where

$$s_k = \sum_{j=0}^{k} \frac{n_j}{n}$$

It can be easily verified that the resulting histogram $p_s(s_k)$ is uniform.

Histogram $p_r(r_k)$ can be modified to $p_z(z_k)$ by performing the following steps.

1. Equalize the source histogram and get the gray levels s_k and the mapping function T. i.e. $T(r_k) = s_k$.

2. Equalize the target histogram and get the gray levels v_k and the mapping function G. i.e. $G(z_k) = v_k$.

3. Find the inverse transformation G^{-1}.

4. Modify the image such that $z_k = G^{-1}[T(r_k)]$.

The resultant histogram is the target histogram.

In Imagetool, we have provided six different density functions to which the image can be modified. They are Gaussian, Uniform, Exponential, Rayleigh, Hyperbolic (Cube root) and Hyperbolic (Logarithmic). These are listed below.

Gaussian

$$p_s(s) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{\frac{(s-\mu)^2}{2\sigma^2}\right\}$$

Uniform

$$p_S(s) = \frac{1}{s_{max} - s_{min}}$$

Exponential

$$p_S(s) = \alpha \exp \{ -\alpha (s - s_{min}) \}$$

Rayleigh

$$p_S(s) = \frac{s - s_{min}}{\alpha^2} \exp \left\{ - \frac{(s - s_{min})^2}{2\alpha^2} \right\}$$

Hyperbolic (Cube root)

$$p_S(s) = \frac{1}{3} \frac{s^{-2/3}}{s_{max}^{1/3} - s_{min}^{1/3}}$$

Hyperbolic (Logarithmic)

$$p_S(s) = \frac{1}{s [\ln s_{max} - \ln s_{min}]}$$

These modification subservices can be availed by using the routines *image_norm*(), *image_uni*(), *image_exp*(), *image_rayl*(), *image_hyp_cbrt*() and *image_hyp_log*().

(III) Point Operations

In practical image processing, one often needs to do various point operations which are computationally cheaper than all the previously described operations but, since the data size is too large, they need to be parallelized.

Such operations are offered by the point operations server. These are *image_neg*(), to find negative of a given picture, *image_add*(), to add two pictures, *image_sub*(), to subtract two pictures and *image_slice*(), to bit-slice a given picture. Image subtraction finds applications in segmentation of moving pictures where edges and other features could be found by subtracting consecutive frames of moving object. Image addition is used to reduce noise and is usually utilized as a low-pass filter. Bit-slicing is another popular operation which separates the pixels according to their particular bit being 0 or 1.

3.2.4 Itool Interface to Iserv

Itool, as mentioned earlier, allows the user to utilize the subservices offered by Imagetool in a user-friendly manner. But in that process, it somewhat restricts the user from the freedom that these functions offer. For example, image processing subservices can be performed in any buffer maintained in the subserver by specifying the buffer number. Through Itool these can be done only on the image stored in buffer 0 and the output image is kept in buffer 1. This has been done in order to keep the interface clean. However, for each subserver, a copy service is provided to copy buffers by specifying the source and the destination buffer. If one wants to perform averaging on buffer 3, one can copy the buffer 3 to 0 and perform that operation.

For the same reason, various parameters required for different subservices are defaulted to certain values which are most commonly used. These parameters can be changed at the installation time.

Since maximum freedom is available to the user while using the library routines for image procesing, we suggest that for writing specific applications, the user should avail the libray functions directly and not through Itool. The complete syntax of the library routines is given in Appendix D.

Chapter 4

Conclusions

4.1 Conclusions

A run-time system for executing ParC programs has been developed to make use of the workstation network environment at Department of Computer Science and Engineering, I. I. T., Kanpur. All the features of ParC like threads, communication through channels, timers, synchronization primitives like semaphores are implemented. The concept of a centralized communication server for handling the problems of synchronization and uniformity is used. The pseudo-busy-waiting strategy for communication is used and was found to be more effective for multi-threaded applications. For ease of distributing the tasks over the network and establishing communication links between them, an interpreter for a configuration language, config, is provided. ParC implementation provides for portability between Sun network and the Transputer networks like Hathi-2 [Aspnas90] essentially because of the source code compatibilty of the ParC programs written on this network with those on the transputer-based systems. Templates for several standard topologies at the task level are also provided so that the applications are configurable to any number of processors. Thus our system is more general than the earlier systems reported in [Das90] & [Ghoshal90] which have fixed topologies and number of processors.

As an illustration of the capability of this parallel platform, a general purpose image processing system is implemented using ParC. This system is designed so as to be easily extensible to include more functions specific to application at hand. Also, it has the provision to be interactive to the user by means of an interactive package as a front end based on Sunview, a popular windowing package on Sun workstations.

4.2 Future work

The ParC platform forms an important part of an ongoing project at I.I.T., Kanpur. The aim of this project is to provide a futuristic environment for parallel programming based on the existing hardware facilities. Following paragraphs describe the scope for extension and ways of

integration of this module into the entire project.

This platform can be extended to run on a heterogenous environment with clever use of underlying network communication facilities and sharing of the file systems. This will facilitate the number crunching required for most scientific applications which are now programmed on traditional sequential computers. Because of the varying computing power of different nodes on the network, the load-balancing will be complicated and different heuristics for load-balancing will have to be applied. Design of a load-balancing algorithm is an interesting research area [Lundberg89]. Also the routines for conversion to and from the different data representations on different machines will have to be included in this platform without losing the simplicity and the elegance which this platform provides.

The efficacy of the pseudo-busy-waiting strategy for a large number of applications can be analysed and the communication server can be modified to take into account the different strategies if the abovementioned is not suitable. The thread-handling mechanism should be modified so as to make it optimal. In this system, we use the Sun lightweight processes and a computationally cheaper alternative like the one reported in [Srikanth91] can be incorporated. Considering the size of source code of applications for simulation of different architectures [Plata90][Zapata90] on this platform, a high-level language for this should be developed and a translator can be written to convert this language to ParC. A debugger for ParC can be provided and can also be equipped with graphical display at the communications server level [Griffin88]. A performance evaluation tool for assessing the efficacy of different architectures for an applications like PAWS [Pease91] or RTS [Qin88] will be useful. Algorithms for shared-memory multiprocessors are difficult to simulate on our system. A shared-memory scheme like tuple-space based Linda [Gelertner85], a global name space [Fleckenstein89] or shared data structures [Sullivan88] can be implemented using ParC [Rizzo89].

The run-time system provided for ParC can be easily extended to other languages as well. For instance FORTRAN can be supplied with the run-time library similar to ParC and this platform can be extended for

programming in different languages similar to MLP [Hayes88]. Cost analysis for different multitasking and synchronization primitives should be done and results compared with [Atkins88]. A parallelizing FORTRAN compiler is developed [Prabhudeva91] and can be modified to incorporate the run-time library extended for FORTRAN. Algorithm degradation can be severe due to communication costs both in terms of resource use and waiting delay [McCreary89]. Hence the system will have to be modified for automatic determination of grain-size. A distributed language DC [Subramaniam91] is implemented on the Sun network and can be included in our mixed language programming system after a few modifications.

As far as the development of applications on this platform is concerned, there is a vast number of applications which can be parallelized and run on the network [Bell89][Pancake90]. In addition to the image processing package Imagetool, different applications of image processing like contextual classifier etc. are being developed. Also, this platform is well-suited for compute-intensive applications which have high compute-to-communication ratio like the finite element methods in engineering.

References

[Andre85] Andre,F., Herman,D. & Verjus,J.P., *Synchronization of parallel programs*, The MIT Press, 1985.

[Aspnas90] Aspnas,M., Back,R.J.R., Malen,T-E, *Hathi-2 multiprocessor system*, Microprocessors and microsystems, Vol. 14(7), September 1990.

[Atkins88] Atkins,M.S. & Olsson,R.A., *Performance of multitasking and synchronization mechanisms in the programming language SR*, Software - Practice & Experience, Vol. 18(9), September 1988, pp 879-895.

[Babb88] Babbll,R.G., *Programming parallel processors*, Addison-Wesley Publishing Company, Reading, Massachusetts ,1988.

[Bal89] Bal,H.E., Steiner,J.G. & Tanenbaum,A.S., *Programming languages for distributed computing systems*, ACM Computing Surveys, Vol. 21(3), September 1989, pp 261-322.

[Bell89] Bell,G., *The future of high performance computers in science and enginnering*, Communications of the ACM, Vol. 32(9), September 1989, pp 1091-1101.

[Bershod88] Bershod,B., Lazowska,E. & Levy,H.M., *PRESTO : A systems for object oriented parallel proramming*, Software - Practice & Experience, Vol. 18(8), August 1988, pp 713-732.

[Bornat86] Bornat,R., *A protocol for generalised occam*, Software - Practice & Experience, Vol. 16(9), September 1986, pp 783-799.

[Brawer89] Brawer,S., *Introduction to Parallel programming*, Academic Press, 1989.

[Cantoni86] Cantoni,V. & Levialdi,S., *Pyramidal systems for computer vision*, Springer Verlag, 1986.

[Cooper88] Cooper,R.E.M. & Jones,G., *A microprogrammed occam interpreter for the HLH Orion*, Software - Practice and Experience, Vol. 18(1), Jan 1988, pp 63-71.

[Das90] Das,S.R., Vaidya,N.H. & Patnaik,L.M., *Design and implementation of a hypercube multiprocessor*, Microprocessors and Microsystems, Vol. 14(2), March 1990, pp 101-106.

[Dougherty87] Dougherty,E.R. & Girdina,C.R., *Matrix structured image processing*, Prentice hall Inc., Englewood cliffs, New Jersey, 1987.

[Dubnicki88] Dubnicki,C., Madey,J. & Wygladala,W., *Edison-N - an Edison implementation for a network of microcomputers*, Software - Practice and Experience, Vol. 18(4), April 1988, pp 349-363.

[Dubois88] Dubois,M., Scheurich,C. & Briggs,F.A., *Synchronization, coherence and event ordering in multiprocessors*, IEEE Computer, Vol. 21(2), February 1988, pp 9-21.

[Fisher86] Fisher,A.J., *A multiprocessor implementation of occam*, Software - Practice & Experience, Vol. 16(10), October 1986, pp 875-892.

[Fleckenstein89] Fleckenstein,C.J. & Hemmedinger,D., *Using a global name space for parallel execution of Unix tools*, Communications of the ACM, Vol. 32(9), September 1989, pp 1085-1090.

[Gammage87] Gammage,N.D., Kamel,R.F. & Casey,L.M., *Remote randezvous*, Software - Practice & Experience, Vol. 17(10), October 1987, pp 741-755.

[Gehani86] Gehani,N.H. & Roome,W.D., *Concurrent C*, Software - Practice & Experience, Vol. 16(9), September 1986, pp 821-844.

[Gehani88] Gehani,N.H., & Roome,W.D., *Concurrent C++ : Concurrent programming with class(es)*, Software - Practice & Experience, Vol. 18(12),

December 1988, pp 1157-1177.

[Gelertner85] Gelertner,D., *Parallel programming in Linda*, Proceedings of the international conference on parallel processing, August 1985, pp. 255-263.

[Ghoshal90] Ghoshal,S.K., Guha,S., Ariff,S.M. & Rajaraman,V., *Simple low-cost multiprocessor based on message-passing FIFO links*, Microprocessors and Microsystems, Vol. 14(5), June 1990, pp 297-300.

[Griffin88] Griffin,J.H., Wasserman,H.J. & McGavran,L.P., *A debugger for parallel processes*, Software - Practice & Experience, Vol. 18(12), December 1988, pp 1179-1190.

[Gonzalez77] Gonzalez,R.C. & Wintz,P., *Digital Image Processing*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1977.

[Hall89] Hall,G., Terrell,T.J., Senior,J.M. & Murphy,L.N., *Transputer-based implementation of the radon transform*, Microprocessors and Microsystems, Vol. 13(7), September 1989, pp 449-455.

[Hamey89] Hamey,L.G.C., Webb,J.A. & Wu,I.C., *An architecture independent programming language for low level vision*, Computer Vision, Graphics and Image Processing, Vol. 48, p. 246-264.

[Hansen87a] Hansen,P.B., *Joyce - A programming language for distributed systems*, Software - Practice & Experience, Vol. 17(1), January 1987, pp 29-50.

[Hansen87b] Hansen,P.B., *A Joyce implementation*, Software - Practice & Experience, Vol. 17(4), April 1987, pp 267-276.

[Hansen89a] Hansen,P.B., *The Joyce language report*, Software - Practice & Experience, Vol. 19(6), June 1989, pp 553-578.

[Hansen89b] Hansen,P.B., *A multiprocessor implementation of Joyce*, Software - Practice & Experience, Vol. 19(6), June 1989, pp 579-592.

[Hayes88] Hayes,R., Manweiler,S.W. & Schlichting,R.D., *A simple system for constructing distributed, mixed language programs*, Software - Practice & Experience, vol. 18(7), July 1988, pp 641-660.

[Hoare78] Hoare,C.A.R., *Communicating sequential processes*, Communications of the ACM, Vol. 21(8), August 1978.

[Hoare89] Hoare,C.A.R., *Communicating sequential processes*, Prentice hall India Pvt. Ltd., New Delhi, 1989.

[Hwang85] Hwang,K. & Briggs,F.A., *Computer architecture and parallel processing*, McGraw Hill, 1985.

[Inmos84] Inmos Ltd., *Occam Programming Manual*, Prentice Hall International, 1984.

[Inmos89] Inmos Ltd., *Parallel C reference Manual*, 1989.

[Jesshope89] Jesshope,C., *Parallel processing, the transputer and the future*, Microprocessors and Microsystems, Vol. 13(1), January 1989, pp. 33-37.

[Kerridge86] Kerridge,J. and Simpson,D., *Communicating parallel processes*, Software - Practice & Experience, Vol. 16(1), January 1986, pp 63-86.

[Landy84] Landy,M.S., Cohen,Y. & Sperling,G., *HIPS : A unix-based image processing system*, Computer Vision, Graphics and Image Processing, Vol. 25, pp. 331-347.

[Logrippo88] Logrippo,L., Obaid,A., Briand,J.P. & Tehri,M.C., *An interpreter for LOTOS, A specification language for distributed systems*, Software - Practice and Experience, Vol. 18(4), April 1988, pp 365-385.

[Lundberg89] Lundberg,L., *A parallel Ada system on an experimental multiprocessor*, Software - Practice & Experience, Vol. 19(8), August 1988, pp 787-800.

[Matsuyama89] Matsuyama,T., *Expert systems for image processing : knowledge based composition of image analysis processes*, Computer Vision, Graphics and Image Processing, Vol. 48, pp. 22-49.

[McCreary89] McCreary,C. & Gill,H., *Automatic determination of grain-size for efficient parallel processing*, Communications of the ACM, Vol. 32(9), September 1989, pp 1073-1078.

[Pancake90] Pancake,C.M. & Bergmark,D., *Do parallel languages respond to the needs of scientific programmers ?*, IEEE Computer, Vol. 23(12), December 1990.

[Pease91] Pease,D. et. al., *PAWS : A performance evaluation tool for parallel computing systems*, IEEE Computer, Vol. 24(1), January 1991, pp 18-30.

[Plata90] Plata,O.G. et. al., *ACLE : A software package for SIMD computer simulation*, The computer journal, Vol. 33(3), pp. 194-203.

[Prabhudeva91] Prabhudeva,H.T., *A parallel fortran translator*, M. Tech. Thesis, Department of Computer science and engineering, I. I. T. Kanpur, March 1991.

[Qin88] Qin,B., Sholl,H.A., Ammar,R.A., *RTS : A system to simulate the realtime cost behaviour of parallel computation*, Software - Practice & Experience, Vol. 18(10), October 1988, pp 967-985.

[Reeves84] Reeves,A.P., *Parallel computer architectures for image processing*, Computer Vision, Graphics and Image Processing, Vol. 25, pp. 68-88.

[Rizzo89] Rizzo,L., *Simulation and performance evaluation of parallel software on multiprocessor system*, Microprocessors and Microsystems, Vol. 13(1), january, 1989, pp 39-46.

[Rosenfeld76] Rosenfeld,A. & Kak,A.C., *Digital Picture Processing*, Academic Press, 1976.

[Shoja88] Shoja,G.C., Clarke,G. & Taylor,T., *REM : A distributed facility for utilizing idle processing power of workstations*, In " Distributed Processing", eds. Barton. M.H. et.al., Elsevier Science Publishers, 1988.

[Skillicorn90] Skillicorn,D.B., *Architecture independent parallel computation*, IEEE Computer, Vol. 23(12), Decemebr 1990, pp 38-49.

[Srikanth91] Srikanth,T., *CPPE : Concurrent and Parallel Programming Environment*, M. Tech. thesis, Department of Computer Science and Engineering, I. I. T., Kanpur, May 1991.

[Srivastava90] Srivastava,A.K, Kshetramade,S. & UdayaBhaskar, B., *Image processing on a network of transputers*, Proceedings of the workshop on parallel processing, B.A.R.C., Bombay, February 1990.

[Subramaniam91] Subramaniam,D.S.M., *Enhancements to DC : A distributed programming language*, M. Tech. Thesis, Department of Computer science and engineering, I. I. T. , kanpur, February 1991.

[Sullivan88] Sullivan,M. & Anderson,D., *Marionette : A system for Parallel distributed programming using a master/slave model*, Tech. Rep., Computer science division, University of California, Berkeley, November 1988.

[Sun88] Sun Microsystems, *Network programming manual*, May, 1988.

[Tamura83] Tamura,H., Sakane,S., Tomita,F. & Yokoya,N., *Design and implementation of SPIDER - A transportable image processing software package*, Computer Vision, Graphics and Image Processing, Vol. 23, pp. 273-294.

[Transputer90a] *Special issue on applying the transputer (part I)*, Microprocessors and Microsystems, Vol. 13(2), March89.

[Transputer90b] *Special issue on applying the transputer (part II)*, Microprocessors and Microsystems, Vol. 13(3), April89.

[Tripathi89] Tripathi,A. & Berge,E., *An implementation of the object-oriented concurrent programming language - SINA*, Software - Practice & Experience, Vol. 19(3), March 1989, pp 235-256.

[Tulshibagwale90] Tulshibagwale,A.V. & Sapre,S.N., *A dynamic message redirection kernel*, Proceedings of the workshop on parallel processing, B.A.R.C, Bombay, February 1990.

[Udupikar91] Udupikar,V. (CDAC, Pune), *ImagePro*, Personal communication , March 1991.

[Vernon88] Vernon,D. & Sandini,G., *VIS : A virtual image system for Image understanding research*, Software - Practice & Experience, Vol. 18(5), May 1988, pp 395-414.

[Walker85] Walker,P., *The Transputer - A building block for parallel processing*, Byte, May 1985.

[Wallace89] Wallace,R.S., Webb,J.A. & Wu,I.C., *Machine independent image processing : performance of Apply on diverse architectures*,

Computer Vision, Graphics and Image Processing, Vol. 48, pp. 265-276.

[Walters87] Walters,D., *Selection of image primitives for general purpose visual processing*, Computer Vision, Graphics and Image Processing, Vol. 37, pp. 261-298.

[Wrench88] Wrench,K.L., *CSP-i : An implementation of CSP*, Software - Practice & Experience, Vol. 18(6), June 1988, pp 545-560.

[Zapata90] Zapata,E.L. et. al., *Software tools for multiprocessor simulation and programming*, Cybernetics and systems: an international journal, Vol. 21, pp. 291-310.

Appendix A

USER MANUAL (RUN-TIME LIBRARY)

1. Introduction

The ParC run-time library consists of the compiled functions which can be used directly by programs. This appendix explains the conventions to call these functions, the arguments and the results of them. These functions are available for calling from a ParC program apart from the normal C functions. For using ParC library, the user has to include the file "uparc.h" in the beginning of the program using the standard C preprocessor statement "#include" as following:

#include "uparc.h"

This file contains all the definitions required for the use of the ParC functions. In addition to this, the user has to compile the program using the ParC compiler "pcc" , the options for which are the same as that of Sun C compiler "cc". The source file name must have the extension ".c". e.g.

pcc -o addsvr addsvr.c -D__DEBUG__ -I.

2. ParC Functions

The ParC functions can be grouped into 4 categories. They are:

1.Creation of various execution threads.

2.The message-passing primitives.

3.Using software semaphores.

4.Using Timer facilities.

Functions in each of these categories are described below.

2.1. Threads

2.1.1. thread_create : *create a simple thread.*

```
ws = thread_create(fn, wssize, nargs, arg1, ..., argn)
char *ws;
void (*fn)();
int wssize;
```

int nargs;

int arg1,arg2,...,argn;

thread_create creates a new thread running function fn at the same priority as that of the invoking thread. The workspace of wssize bytes is taken from the heap and the pointer to it is returned by this function. After the thread finishes execution, this workspace is returned to the heap. The nargs arguments arg1,arg2,...,argn are passed to the function fn. If no space is available for the creation of new thread, thread_create returns a NULL pointer. For wssize not divisible by wordsize, (which is 4 on Sun systems) , workspace of size which is next multiple of wordsize is allotted. Another function thread_start allows user to specify his own workspace.

2.1.2.thread_deschedule:*make thread unable to continue temporarily.*

result= thread_deschedule();

int result;

This routine causes the invoking thread to be suspended until it gets scheduled again for execution. When a thread is suspended , the next thread on the scheduler's list gets the processor time slot. This function can be used by threads which are busy waiting on some condition to be satisfied, in this way it would not unnecessarily take the processor time which can be used by some other thread to perform its computation. It will return -1 if the routine fails due to some internal error.

2.1.3. thread_priority : *return thread's priority of execution.*

prio = thread_priority()

int prio;

A thread in ParC may have any of the two priorities:

1.THREAD_URGENT

2.THREAD_NOTURG

The scheduler maintains separate lists for the threads running with these priorities. Unless any THREAD_URGENT thread is running, the scheduler selects one of the THREAD_NOTURG thread for execution which

will continue till either an URGENT thread becomes ready to run or the time slot for current thread gets over. This function returns the priority of the invoking thread.

2.1.4. thread_restart : restart a given thread

result= thread_restart(p)

int result;

*char *p;*

This routine can be used for restarting a thread which was stopped earlier because of a call to chan_reset while it was trying to communicate over a channel. The thread will be restarted from where it left off. That is , it will again try to communicate over the same channel. p is a pointer to the workspace of the thread.

2.1.5. thread_start : *a general thread creation function*

result=thread_start(fn,ws,wssize,flags,nargs,arg1,...,argn)

int result;

*void (*fn)();*

*char *ws;*

int wssize;

int flags;

int nargs;

int arg1,...,argn;

This function creates a new thread running the function fn. This is more general than thread_create because the user may specify the workspace ws of the size wssize that the new thread is supposed to use. Also the priority with which the new thread is to be invoked can be specified as flags. flags can be THREAD_URGENT or THREAD_NOTURG. nargs arguments arg1,..argn are passed to the the function fn. Since only integer arguments arg1,..argn can be passed to the thread with thread_create, this function gives additional advantage of passing structures as value parameters by copying them into the workspace in the required format and making the new thread use that workspace. It will return (-1) if the thread can not be created because of any error.

2.1.6. **thread_stop** : stop the current thread

thread_stop()

This function stops the invoking thread. The workspace used by the thread is released if it was created on heap by thread_create. The current thread is also stopped when its main function returns.

2.2. Channels

The communication through channels in ParC is synchronous. The thread invoking this routine will be suspended until the data becomes available on the channel. Also the access to the channels is exclusive. Thus out of the threads trying to use the same channel, only one will succeed and the others will be suspended till the channel is in use. This interface for using channels is common to both intertask and intratask communication. Naturally the intertask communication is a little slower than the intratask one.

2.2.1. **chan_in_byte** : input a byte from a channel

result= chan_in_byte(b,chan)

int result;

*char *b;*

*CHAN *chan;*

This function reads a single byte from the channel pointed to by chan into the character pointed to by b. It returns (-1) if some internal error occurs due to uninitialized channel or illegal channel specification.

2.2.2.**chan_in_byte_t**: *input a byte from a channel or timeout.*

result= chan_in_byte_t(b,chan,timeout)

int result;

*char *b;*

*CHAN *chan;*

int timeout;

When this function is invoked a request to input a character from the channel pointed to by chan is issued. If the communication does not

take place within timeout ticks of the timer, this function returns with value 0, cancelling the input request. If data is available on the specified channel then a positive value is returned and the input request is satisfied. return value of (-1) indicates some internal error. The default value for one tick of timer is 1 microsecond.

2.2.3. **chan_in_word** : input a word from a channel

result= chan_in_word(w,chan)
int result;
*int *w;*
*CHAN *chan;*

This function reads a single integer from the channel pointed to by chan into the integer pointed to by w. It returns (-1) if some internal error occurs due to uninitialized channel or illegal channel specification. Error can also occur because of not enough data being available. (i.e. available data on the channel is less than wordsize in length.)

2.2.4.chan_in_word_t: *input a word from a channel or timeout.*

result= chan_in_word_t(w,chan,timeout)
int result;
*int *w;*
*CHAN *chan;*
int timeout;

When this function is invoked a request to input an integer from the channel pointed to by chan is issued. If the communication does not take place within timeout ticks of the timer, this function returns with value 0, cancelling the input request. If data is available on the specified channel then a positive value is returned and the input request is satisfied. return value of (-1) indicates some internal error. The default value for one tick of timer is 1 microsecond.

2.2.5. **chan_in_message** : *input a message from a channel*

result= chan_in_message(l,b,chan)
int result;
int l;

*char *b;*

*CHAN *chan;*

This function reads l bytes from the channel pointed to by chan into the character array pointed to by b. It returns (-1) if some internal error occurs due to uninitialized channel or illegal channel specification. Error also occurs if the data available on the channel is not sufficient.

2.2.6.chan_in_message_t:*input a message from a channel or timeout.*

result= chan_in_message_t(l,b,chan,timeout)

int result;

int l;

*char *b;*

*CHAN *chan;*

int timeout;

When this function is invoked a request to input a message of length l from the channel pointed to by chan is issued. If the communication does not take place within timeout ticks of the timer, this function returns with value 0, cancelling the input request. If data is available on the specified channel then a positive value is returned and the input request is satisfied. return value of (-1) indicates some internal error. The default value for one tick of timer is 1 microsecond.

2.2.7. chan_init : *initialize a channel*

result= chan_init(chan)

int result;

*CHAN *chan;*

This function initializes the channel word pointed to by chan. This clears any communication pending on the channel, waking up any thread waiting on it with error. All channels declared in the program must be initialized before attempting to communicate using them. The channels used for interprocessor communication are already initialized and need not be initialized using this routine.

2.2.8. chan_out_byte : *output a byte to a channel*

result= chan_out_byte(b,chan)

int result;

*char *b;*

*CHAN *chan;*

This function writes a single character pointed to by b onto the channel pointed to by chan. It returns (-1) if some internal error occurs due to uninitialized channel or illegal channel specification.

2.2.9. chan_out_byte_t : *output a byte to a channel or timeout.*

result= chan_out_byte_t(b,chan,timeout)

int result;

*char *b;*

*CHAN *chan;*

int timeout;

When this function is invoked a request to output a character onto the channel pointed to by chan is issued. If the communication does not take place within timeout ticks of the timer, this function returns with value 0, cancelling the output request. If data is requested on the specified channel then a positive value is returned and the output request is satisfied. return value of (-1) indicates some internal error. The default value for one tick of timer is 1 microsecond.

2.2.10. chan_out_word : *output a word to a channel*

result= chan_out_word(w,chan)

int result;

*int *w;*

*CHAN *chan;*

This function writes a single integer onto the channel pointed to by chan from the integer pointed to by w. It returns (-1) if some internal error occurs due to uninitialized channel or illegal channel specification. Error can also occur because of not enough data being requested. (i.e. requested data on the channel is less than wordsize in length.)

2.2.11. chan_out_word_t : *output a word to a channel or timeout.*

result= chan_out_word_t(w,chan,timeout)

int result;

*int *w;*
*CHAN *chan;*
int timeout;

When this function is invoked a request to output an integer onto the channel pointed to by chan is issued. If the communication does not take place within timeout ticks of the timer, this function returns with value 0, cancelling the output request. If data is requested on the specified channel then a positive value is returned and the output request is satisfied. return value of (-1) indicates some internal error. The default value for one tick of timer is 1 microsecond.

2.2.12. **chan_out_message** : *output a message to a channel*

result= chan_out_message(l,b,chan)
int result;
int l;
*char *b;*
*CHAN *chan;*

This function writes a message of length l onto the channel pointed to by chan from the character array pointed to by b. It returns (-1) if some internal error occurs due to uninitialized channel or illegal channel specification. Error can also occur because of not enough data being requested. (i.e. requested data on the channel is less than l in length.)

2.2.13. **chan_out_message_t** : *output a message to a channel or timeout.*

result= chan_out_message_t(l,b,chan,timeout)
int result;
int l;
*char *b;*
*CHAN *chan;*
int timeout;

When this function is invoked a request to output a message b of length l onto the channel pointed to by chan is issued. If the communication does not take place within timeout ticks of the timer, this function returns

with value 0, cancelling the output request. If data is requested on the specified channel then a positive value is returned and the output request is satisfied. return value of (-1) indicates some internal error. The default value for one tick of timer is 1 microsecond.

2.2.14. chan_reset : *reset a channel*

```
ws = chan_reset(chan)
char *ws;
CHAN *chan;
```

This function resets the channel pointed to by chan. If a thread was attempting to communicate at the time of resetting, this function returns the workspace pointer to that thread, using which that thread can be restarted. If a channel is idle at the time of calling this function, it returns a special constant NotProcess_P which is the indication of an idle channel.

2.3. Semaphores

Semaphores are provided in ParC in order to achieve synchronization between threads and also for mutual exclusion in accessing some shared resource. A global semaphore , par_sema ,for accessing the system resources such as display is provided. When multiple threads try to write onto the screen, the output may be garbled. This problem can be taken care of by each thread following a protocol of waiting on par_sema, writing on screen and signalling par_sema. Except for par_sema, all other semaphores should be accessed by the threads of the same priority. For example, it will not be acceptable to synchronize an urgent and a non-urgent thread using semaphores. It has to be done using channel I/O of a NULL message.

2.3.1. sema_init : *initialize a semaphore*

```
result= sema_init(s,v)
int result;
SEMA *s;
int v;
```

This function initializes the semaphore pointed to by s. This

involves emptying the queue of threads waiting for the semaphore and making the value of semaphore equal to v. If a static or extern semaphore is uninitialized, it has an empty wait-queue and the initial value of 0. Whereas the auto semaphore, if uninitialized, may contain unpredictable data and may cause the system to crash if waited upon by some thread. This function returns (-1) if some internal error occurs, 0 otherwise.

2.3.2. sema_signal : *perform signal operation on a semaphore*

result= sema_signal(s)
int result;
*SEMA *s;*

Invoking this function will cause on of the threads waiting for the semaphore pointed to by s to wake up and execute again resetting the value of s. If there are no threads waiting on s , the value of the semaphore will be incremented by 1. Return value is (-1) if some error occurs, 0 otherwise.

2.3.3. sema_signal_n : *perform n signal operation on a semaphore*

result= sema_signal_n(s,n)
int result;
*SEMA *s;*
int n;

This function is equivalent to calling sema_signal n times in a single time slice. It will return (-1) if an error occurs, 0 otherwise.

2.3.4. sema_wait : *perform a wait operation on a semaphore.*

result= sema_wait(s)
int result;
*SEMA *s;*

When a thread calls this function, it will be put in the wait-queue of semaphore pointed to by s if the value of the semaphore is 0. If value of the semaphore is nonzero, it will return decreasing the value. When a thread is put in the wait-queue it will be resumed only when some other thread does sema_signal. Programs should not rely on the order in which the threads are woken up. At present the FIFO order is maintained. This function returns (-1) in case of error, 0 otherwise.

2.3.5. **sema_wait_n** : *perform n wait operation on a semaphore.*

result= sema_wait_n(s,n)

int result;

*SEMA *s;*

int n;

This operation is equivalent to doing n sema_wait operations in a single time slice. It will return (-1) if an error occurs internally, 0 otherwise.

2.4. Timer

2.4.1. **timer_after** : *compare two timer values*

result= timer_after(t1,t2)

int result;

int t1;

int t2;

This function returns TRUE if timer value t1 is after timer value t2 and FALSE otherwise. Direct comparison between two timer values should be avoided for portability reasons.

2.4.2. **timer_delay** : *delay for some number of ticks*

result= timer_delay(d)

int result;

int d;

This function causes the invoking thread to be delayed by at least d ticks of the timer associated with the priority of that thread. The default value of one tick of timer is one microsecond.

2.4.3. **timer_now** : *return the current timer value*

t = timer_now()

int t;

This function returns the current value of the timer associated with the priority of the invoking thread.

2.4.4. **timer_wait** : *wait until timer reaches given value.*

result= timer_wait(t)

int result;

int t;

The thread calling this function is suspended until the timer associated with the priority of that thread reaches at least t. This function returns 0 if no error occurs, (-1) otherwise.

Appendix B

USER MANUAL (Configuration Language)

1. Introduction

The configuration language which is interpreted by the config program on the Sun network is exactly the same as the general purpose configuration language on transputer-based systems. This language frees user from the botheration of distributing his task over the network and makes the ParC application independent of the topology of the network. Using config one can define his own network topology, establish connections between various tasks, initialize the basic intertask communication mechanism i.e. channels. The config program thus makes the application portable on different networks. This appendix describes the configuration language in detail.

2. Using config

config is the interpreter for the configuration language. An application which consists of multiple tasks to be run on the network is described using this language in a file, the name of which is ending with ".cf" . Then at the command line, type config <file>.cf If the user wants the diagnostic output to be generated, the config programs is run with a "-t" option. This output is contained in a file <file>.cf.tbl. The diagnostic output consists of the tables of processor information, actual sites, the tasks information and the connections. It also contains the number of processors and tasks mapped onto each actual site. Future versions plan to include the time taken and the efficiency of each site so as to serve as a history for further development. When the programs written in configuration language have any errors, the config programs stops at the line containing error and does not generate any process on the remote machine. The error messages which config generates are self explanatory. The defaults for config are as follows:

Maximum number of sites : 16

Maximum number of processor declarations : 128

and

processor host !This is also a comment.

3.2. Continuation

In config , each statement must end with a newline character. If a statement has to continue for more than one line, then the last character of the incomplete line should be a hyphen "-". Continuation and comments can be used together. e.g.

PROCESSOR - !combination of comment and continuation.

host

3.3. Constants

Different types of constants are available in config for various purposes. The two major types are numeric and string constants.

3.3.1. Numeric Constants

Numeric constants are either decimal or hexadecimal. The decimal constants can be integers or real numbers. Further decimal constants can be appended with a scaling letter "K" or "M" to denote scaling by 1024 and (1024X1024) respectively. These are very useful in specifying the memory requirements etc. (This feature is not used in this version but the scaling constants are still recognized for compatibility reasons.) The detailed syntax for constants follows:

constant : decimal_constant | hex_constant ;

hex_constant : "&"hex_digits ;

hex_digits : (hex_digit)+ ;

hex_digit : digit | [A-F] | [a-f] ;

digit : [0-9] ;

decimal_constant : decimal_digits ["." (decimal_digits)] [scaling_letter] ;

decimal_digits : (digit)+ ;

scaling_letter : "K" | "M" ;

Scaling letter can not be attached to hexadecimal constants. Also for portability reasons, it is suggested that the number of digits after the decimal point in the real decimal constant should be one only.

3.3.2. String Constants

String constants are provided in config to specify the code files in the "TASK" statements. The string consists of a sequence of characters enclosed within a pair of double quotes. The string can not contain a double quote or a newline. If newline is immediately following the string, the last double quote can be excluded. User should take care that the filename should not contain a double quote because there is no way to specify that file to config.

3.4. Identifiers

The config application consists of a number of objects like processors, tasks, connections and wires. Each of these objects can be given a unique identifier. These identifiers are used both for identification of the objects and error reporting. Identifiers in config are the sequences of alphanumeric constants and special characters like an underscore "_" and a dollar sign "$". The difference in case does not matter. e.g.

processor some_processor_name_as_illustration$

and

processor some_PROcessor_name_AS_Illustration$

amount to the same declaration. Sometimes it is not necessary to give any name to an object. In that case, the user can simply put a question mark "?" in place of the identifier. e.g.

connect ? addsvr[0] addclnt[2]

Since it is not required to refer to this connection by name in the config program, one may skip the identifier as shown above. It follows that skipping the identifier by putting the question mark does not allow one to refer to that object.

3.5. PROCESSOR statement

The processor statement allows user to declare a logical processor. In the Inmos ParC, there are declarations for only physical processors using this statement which impaires the portability across the networks. Thus config on Sun provides user with a superset introducing the concept of a logical processor. A number of logical processors are mapped onto the physical sites available at run-time. Thus if any of the sites is not running, the application is still unchanged unlike the original ParC. Also config takes care that all the tasks which are supposed to run on the same logical processor are on the same physical processor. All the processors need to be declared before any reference to them. Otherwise config will stop and give the error message "processsor XXXXX not declared!." The processors declared can be referenced in the "WIRE" as well as "PLACE" statements. For compatibility reasons config accepts the processor attribute component which can be of the form "TYPE = PC". Also a special processor called host has to be declared. The "host" processor is always mapped onto the actual site from which the command to run the application is given. The syntax of the PROCESSOR statement follows:

processor_statement : *"PROCESSOR"* [*identifier* | *"?"*] [*processor_attribute*] ;

processor_attribute : *"TYPE" "=" processor_type* ;

processor_type : *"PC"* ;

Each processor in the network is supposed to have four links. These links are used for physical connections between these processors and are referred to with a link specifier. The syntax of the link specifier is as follows :

link_specifier : *processor_id "[" constant "]"* ;

This link specifier is used in WIRE statement.

3.6. WIRE statement

The WIRE statement declares a physical connection between two physical processors. The statement has a syntax like wire_statement :

"WIRE" (identifier | "?") link_specifier link_specifier ;

Since a wire is a bidirectional connection, the order of the link specifier does not matter. Note : On the Sun workstations, we are not concerned about physical connections and thus, this statement is of no consequence but it is included just for compatibility reasons.

3.7. TASK statement

The TASK statement is perhaps the most important statement in config language. It declares a task constituting the application. Each task has a number of attributes describing the task. The task attributes are described below.

3.7.1. INS

Each task must have this attribute giving the number of input channels for the task. If the task does not have any input channel, the INS attribute must be 0. The maximum number of input channels per task is limited to 64.

3.7.2. OUTS

The OUTS attribute specifies the number of output channels for that task. If the task does not have any output channel it should be 0. The maximum number of output channels per task is 64.

3.7.3. FILE

This attribute specifies the filename of the ParC program file. This file must be compiled with the ParC compiler "pcc". If the file does not exist in the directory from which config is executed, it searches for the file according to the PATH variable of the shell. If it is not found in PATH, config stops and reports the error. The FILE attribute may be skipped if the filename is the same as the identifier for that task. e.g.

TASK addsvr INS=32 outs=32

will search for the file "addsvr" since no explicit file is specified. Whereas

TASK addsvr INS=32 OUTS=32 FILE="addserver"

will make config search for the file "addserver" instead of "addsvr".

3.7.4. URGENT

If the urgent attribute is given in the task statement, the task's initial thread will be started at the urgent priority level. The default is THREAD_NOTURG.

3.7.5. Other attributes

Other attributes like the memory size attributes and the OPT attributes are specific to PC-based systems and they are ignored on the Sun implementation. The syntax of the TASK statement follows:

task_statement : *"TASK" identifier attribute_list* ;

attribute_list : (*attribute*)+

attribute : *"INS" "=" constant*
| *"OUTS" "=" constant*
| *"FILE" "=" string*
| *"URGENT"*
| *"OPT" "=" opt_area*
| *memory_area "=" memory_amount*;

opt_area : *memory_area* | *"CODE"* ;

memory_area : *"STACK"* | *HEAP"* | *"STATIC"* | *"DATA"* ;

memory_amount : *constant* | *"?"* ;

The channels associated with a particular task can be referred to by chan_specifier. The syntax of the chan_specifier is

chan_specifier : *task_id "[" constant "]"* ;

The chan_specifier is used for connecting the output channel of a particular task to the input channel of another task using CONNECT statement.

3.8. CONNECT statement

The connect statement connects the output channel of one task to the input channel of another task. The connection is asymmetric and

therefore the order in which these channels are specified is significant. The syntax of the connect statement is as follows:

connect_statement : *"CONNECT" (identifier | "?") out_port in_port ;*

out_port : *port_specifier ;*

in_port : *port_specifier ;*

e.g.

connect ? addsvr[0] addclnt[0]

connects the 0'th output channel of the task "addsvr" to the 0'th input channel of the task "addclnt".

3.9. PLACE statement

This statement forces the config program to map a particular task on a particular processor. This is essential for the tasks which communicate with the user. Such tasks need to be put on the same machine from which the config program is run so that the user can interact with it. This can be achieved by

place interactive_task host

The syntax of the place statement is

place_statement : *"PLACE" task_id proc_id ;*

task_id : *identifier ;*

proc_id : *identifier ;*

Note that before using the PLACE statement, both the task_id and proc_id must be defined using PROCESSOR and TASK statements respectively.

3.9.1. BIND statement

The bind statement is used for allowing the task access to transputer's external event mechanism by binding a channel to some physical port. Even if this is not used on Sun network, it is recognized for compatibility with the transputer-based systems. The syntax of this statement is given below :

bind_statement : *"BIND" binding_type chan_specifier binding_value ;*

binding_type : *"INPUT"* | *"OUTPUT"* ;

binding_value : *"VALUE"* *"="* *constant* ;

4. Error Handling

config utility reports errors in syntax as well as any inconsistencies. In case of error, it immediately stops reading further and reports it. Its error messages are self explanatory and easy to understand. If certain keyword is missing, it reports "XXXXX missing." If any other word takes the place of a keyword, it reports "XXXXX unexpected." It can detect if a program tries to have more number of channels than the maximum allowed. Also if one channel of a task is connected to more than one channel, it reports an error. This program warns if some site is not running. Thus the user can abandon the application if he thinks that because of the inability of that site to function will hamper the performance.

Appendix C

Sun Lightweight Processes

1. Introduction

The Sun lightweight processes mechanism allows user to have different threads of execution sharing the same address space. The "lwp" library provides various routines to manipulate these threads. This appendix describes the Lightweight processes in detail. (Henceforth the Lightweight processes will be simply called "lwps".)

2. Threads

Threads are the independent execution entities which share the code and data space of the process. Each thread has its own stack in order to allow it to call functions independently of each other. Also, since the local variables of a function are created on stack, having a different stack for each thread provides it with the access of its local variables without interfering with other threads. In the lwp mechanism provided by Sun, there is no hierarchy for threads. i.e. there is no parent-child relationship between the created thread and its creator. All the threads which share the address space are grouped together and this is called a pod. The lwp library is implemented on user level and does not have any kernel support. Therefore a system call made by a thread blocks the whole pod. This can be avoided by linking the non-blocking I/O library with the programs using lwp library.

2.1. Identification and Priorities

Each thread has a unique id of the type thread_t. This id is used in order to refer to a thread. A list of threads created is maintained. It is grouped according to the priorities of individual threads. The priority of a thread is used in scheduling. It is a positive number not more than 255 by default. Funotions are provided to change the thread's priority, to know the maximum number of priority levels, to change this number etc. The thread list is divided further into active and inactive threads. A thread is inactive if it is waiting for certain event or is created by its parent in that state. If it is active, it implies that if scheduled, it is ready to run. SELF is a special identifier which specifies the calling thread itself.

2.2. Creation and Destruction

A thread can create another thread by making a call to lwp_create in which it specifies the function to be invoked by the new thread, the arguments to be passed , the stack which the new thread is supposed to use, priority of the created thread and some flags indicating the state in which the thread is to be created. It follows that unless the created thread is of a higher priority than its parent, the parent will return immediately after creating its child. Initially, the first call to any routine in the lwp library, main is transparently converted to a lightweight process. It uses the original stack whereas all the threads created after main will have to allocate space from the heap for their stacks. A thread exits if it specifically calls lwp_destroy(SELF) or any other thread calls lwp_destroy(tid) or the function it is supposed to execute returns. The program exits when all the threads within it (including main) exit. When a thread is destroyed, it frees all the resources consumed by it e.g. stack and the part of heap. The system also transparently creates a number of threads which are used for scheduling , stack management etc. A call to enumerate will show these threads in the thread list. But exiting of the pod does not depend on these system-created threads. A pod exits if only the system-created threads are running.

2.3. Scheduling

The scheduling in Sun lwps is very simple and efficient. The scheduler searches for an active thread with the maximum priority. This thread is made to execute until it relinquishes control or waits for certain event to occur or any higher priority thread becomes ready for execution. Each thread has access to the state of every other thread in the same pod. That is given the id, a thread can know the status of any thread including itself. Each thread can control the scheduling by calling the functions provided by the lwp library. Thus it can reshuffle the order of threads within a priority level, it can put any thread including itself to sleep for certain time. Also it can suspend a thread so that only a call to resume that thread will make it active. A thread can also wait till other thread finishes its job. Handling these facilities without a certain

discipline can cause deadlock or starvation, therefore a programmer should use these functions if absolutely necessary.

2.4. Stack Management

Functions are also provided in the lwp library for stack management. Using these a thread can create new stacks from a cache of stacks maintained by the system. Also an already existing dataspace can be converted to the stack. This is useful if structures of arbitrary sizes have to be passed to threads as parameters. Some software protection is provided by the stack management for the thread to ensure that a thread does not access other thread's stack accidentally or maliciously.

2.5. Context switching

For controlling the context switching, the lwp library provides a number of functions. By default on the context switch, only machine registers are saved in order to provide each thread a virtual machine making it possible to run independently. But normally some other software resources are also used within threads. e.g. the external variable errno is used by all the threads making system calls. Therefore for transparency, this variable should also be included in the context. Using the functions mentioned above, this can be achieved.

2.6. Summary of functions for lwp management

The list of the function related to the lwp handling is given below.

2.6.1. Thread Creation

lwp_self(tid)

lwp_getstate(tid,statevec)

lwp_setregs(tid,machstate)

lwp_getregs(tid,machstate)

lwp_ping(tid)

lwp_create(tid,pc,prio,flags,stack,nargs,arg1,..,argn)

lwp_destroy(tid)

lwp_enumerate(vec,maxsize)

pod_setexit(status)

pod_getexit()

SAMETHREAD(tid1,tid2)

2.6.2. Thread Scheduling

pod_setmaxpri(maxprio)

pod_getmaxpri()

pod_getmaxsize()

lwp_resched(prio)

lwp_setpri(tid,prio)

lwp_sleep(timeout)

lwp_suspend(tid)

lwp_resume(tid)

lwp_yield(tid)

lwp_join(tid)

2.6.3. Error Handling

lwp_geterr()

lwp_perror(s)

lwp_errstr()

2.6.4. Context handling

lwp_ctxinit(tid,cookie)

lwp_ctxremove(tid,cookie)

lwp_ctxset(save,restore,ctxsize,optimise)

lwp_ctxmemget(mem,tid,ctx)

lwp_ctxmemset(mem,tid,ctx)

lwp_fpset(tid)

lwp_liboset(tid)

2.6.5. Stack Management

CHECK(location,result)

lwp_setstkcache(minsize,numstacks)

lwp_newstk()

lwp_datastk(data,size,addr)

lwp_stkcswset(tid,limit)

lwp_checkstkset(tid,limit)

STKTOP(s)

3. Synchronization

For concurrent programming, synchronization of various threads of execution plays an important part. It is this feature that enables the sharing of data, mutual exclusion and reduces the programming efforts. Sun lightweight processes provide two mechanisms of thread synchronization. They are monitors and condition variables. These two provisions actually separates the two main purposes of using another more general synchronization mechanism, the semaphores. The usage of monitors and condition variables in Sun lwp library is described below.

3.1. Monitors

Monitors are used to synchronize the access to shared resources in a concurrent program , by providing for the mutual exclusion mechanism. Though by knowing the scheduling strategy of the lwp scheduler, synchronization can be achieved without using monitors, they are a much more general way of achieving mutual exclusion. In lwp, the monitors are declared to be of the type mon_t. A thread creates a monitor by a call to mon_create and can destroy it with mon_destroy(). When a monitor is created, no thread is "inside" it and the first to "enter" the monitor will always succeed. When a monitor is destroyed, the space allocated to it is freed and the thread waiting for it to become free returns with an error condition. Also all the condition variables bound to a monitor are destroyed with it. The monitor can still exist if the thread which has created it is destroyed. A thread can enter the monitor with a call to mon_enter. within a monitor, it can be entered more than once and the nesting level is maintained so that it decreases with successive calls to mon_exit. When the nesting level becomes zero, the thread exits from the monitor. A function, mon_break, is provided to break the lock of the monitor not necessarily held by the calling thread. This is useful feature for breaking deadlocks, though it allows the undisciplined use of monitors. A thread can enumerate all the monitors, can examine the waiting threads on the monitors and also check if the monitor is currently held by making

appropriate procedure calls in the lwp library. A thread can also conditionally enter the monitor if is not busy. A macro MONITOR is provided for declaring the procedure to be a monitor. The list of all monitor-handling functions follows:

mon_create(mid)

mon_destroy(mid)

mon_enter(mid)

mon_exit(mid)

mon_enumerate(vec,maxsize)

mon_waiters(mid,owner,vec,maxsize)

mon_cond_enter(mid)

mon_break(mid)

MONITOR(mid)

SAMEMON(mid1,mid2)

3.2. Condition Variables

Condition variables are used for synchronization within monitors.It provides a way for waiting on certain condition which might become true due to some other thread's actions. The condition variables can only be used inside the monitors and naturally, each condition variable is bound to certain monitor. In the lwp library they are to be declared of the type cv_t. Condition variables can be created by making a call to cv_create and can be destroyed with cv_destroy. Destroying the monitor to which a condition variable is bound, destroys that condition variable also. When a thread waits on a condition variable cv, it locks cv if it is not engaged and returns immediately. If cv is busy, the thread releases its most recently held monitor, and joins the queue for that condition variable. The queues are maintained according to the scheduling priorities. cv_notify is used for waking up at most one thread waiting for the condition variable. If all the threads blocked for the condition variable are to be awakened, another function, cv_broadcast is provided. A particular thread specified by its id can also be awakened by calling cv_send, if it is blocked on that condition variable. Functions are provided to enumerate the condition variables and to list the blocked threads on a particular condition

variable. It should be noted that after a thread blocked on a condition variable is awakened, it queues up for getting control of the monitor it had released in cv_wait. This situation might cause starvation in some cases. Therefore, instead of cv_notify, a call to cv_broadcast must be used in such condition. The list of functions used for managing condition variables is given below:

cv_create(cv,mid)

cv_destroy(cv)

cv_wait(cv)

cv_notify(cv)

cv_send(cv,tid)

cv_broadcast(cv)

cv_enumerate(vec,maxsize)

cv_waiters(cv,vec,maxsize)

SAMECV(cv1,cv2)

4. Messages

During the course of their execution, threads need to communicate between each other. This need is satisfied in the lwp library by providing routines to send and receive messages. For each thread, a queue of messages is maintained. A message can be sent to a particular thread or all the threads so that any thread can pick them up. The message passing is synchronous. The sender waits till the message is received by other thread and a reply is sent. A receiver waits till the message is sent by the other thread, it has to send a reply to reactivate the sending thread. In a call to msg_send a message buffer and a reply buffer is specified alongwith the identifier of the thread to which the message is sent. Similarly with msg_recv, a buffer has to be specified. A receiver can also specify a timeout which can enable it to regain control if the message does not come within that time. If timeout is INFINITY, the receiver waits infinitely. Reply can be sent using msg_reply. Functions are provided for listing all the threads on both receive and send. The list of message-handling functions is given below:

msg_send(dest,arg,argsize,res,ressize)

msg_recv(sender,arg,argsize,res,ressize,timeout)

msg_reply(sender)

msg_enumsend(vec,maxsize)

msg_enumrecv(vec,maxsize)

MSG_RECVALL(sender,arg,argsize,res,ressize,timeout)

5. Events and Exception handling

Events are the unix signals. Since a number of signals are handled by the lwp library, it is recommended that programs using lwps should handle signals using the special signal handling facilities provided by lwp library. Using functions in the lwp library, one creates the agents which are like threads which generate messages when a signal comes. These messages can be received with msg_recv. Also the enumeration of all the agents is possible with a call to agt_enumerate.

Unlike signals which are the asynchronous events, exceptions can be raised by threads and can be handled using the functions in lwp library. It is similar to , but is more general form of, non-local goto or longjump. The threads can define their own exceptions and can cause the control to be transferred to another thread using these. This can be done by binding a pattern , which is an integer, to an exception handler. Exception is raised by specifying the pattern to exc_raise. There are special functions provided for specifying exception handlers which take care of the exceptions raised on exit of a thread. Event handling and exception handling can be combined so that on receiving a signal, the agent raises an exception instead of sending a message. Though exception handling is the most efficient way of synchronization, it should not be used without certain discipline because it makes the program difficult to debug. The functions achieving event and exception handling are listed below:

agt_create(agt,event,memory)

agt_enumerate(vec,maxsize)

agt_trap(event)

exc_handle(pattern,func,arg)

exc_raise(pattern)

exc_unhandle()

exc_bound(pattern,arg)

exc_notify(pattern)

exc_on_exit(func,arg)

Appendix D

Image Processing Library (l l i b)

The image processing library functions are stored in *ilib.o* file. This file should be linked alongwith the application if it uses any of the functions listed below. The complete syntax and calling convention of each image processing function is listed below. In each of the following functions *fd* is the socket identifier returned by image_init().

Function: *image_init()*

Syntax:

```
int image_init()
```

Description:

Initializes connection to image server (iserv).

Return Value:

Socket identifier to be used for other calls.

-1 if can not create socket.

Function: *image_exit()*

Syntax:

```
int image_exit(sock)
int sock;
```

Description:

Closes connection to image server. *sock* is the socket identifier returned by earlier call to *image_init()*.

Return Value:

-1 on error.

0 otherwise.

Function: *image_con_load()*

Subserver: con

Syntax:

```
int image_con_load(fd , ifile , buf)
int fd , buf;
Ifile ifile;
```

Description:

Loads ifile into buffer number buf in the memory of con subserver. Any existing data in the buffer is overwritten.

Return Value:

0 on error.

1 otherwise.

Function: *image_con_store*()

Subserver: con

Syntax:

```
int image_con_store(fd , ofile , buf);
int fd , buf;
Ifile ofile;
```

Description:

Stores the image in buffer number buf of the con subserver into the file ofile. If the file exists, it is overwritten. The existing image in buf continues to exist.

Return Value:

0 on error.

1 otherwise.

Function: *image_con_copy*()

Subserver: con

Syntax:

```
int image_con_copy(fd , ibuf , obuf);
int fd , ibuf , obuf;
```

Description:

Copies buffer *ibuf* into buffer *obuf* in the con subserver.

Return Value:

0 on error.

1 otherwise.

Function: *image_pnt_load*()

Subserver: pnt

Syntax:

int image_pnt_load(fd , ifile , buf)

int fd , buf;

Ifile ifile;

Description:

Loads ifile into buffer number buf in the memory of pnt subserver. Any existing data in the buffer is overwritten.

Return Value:

0 on error.

1 otherwise.

Function: *image_pnt_store()*

Subserver: pnt

Syntax:

int image_pnt_store(fd , ofile , buf);

int fd , buf;

Ifile ofile;

Description:

Stores the image in buffer number buf of the pnt subserver into the file ofile. If the file exists, it is overwritten. The existing image in buf continues to exist.

Return Value:

0 on error.

1 otherwise.

Function: *image_pnt_copy()*

Subserver: pnt

Syntax:

int image_pnt_copy(fd , ibuf , obuf);

int fd , ibuf , obuf;

Description:

Copies buffer *ibuf* into buffer *obuf* in the pnt subserver.

Return Value:

0 on error.

1 otherwise.

Function: *image_mod_load*()

Subserver: modhist

Syntax:

```
int image_mod_load(fd , ifile , buf)
int fd , buf;
Ifile ifile;
```

Description:

Loads ifile into buffer number buf in the memory of modhist subserver. Any existing data in the buffer is overwritten.

Return Value:

0 on error.

1 otherwise.

Function: *image_mod_store*()

Subserver: modhist

Syntax:

```
int image_mod_store(fd , ofile , buf);
int fd , buf;
Ifile ofile;
```

Description:

Stores the image in buffer number buf of the modhist subserver into the file ofile. If the file exists, it is overwritten. The existing image in buf continues to exist.

Return Value:

0 on error.

1 otherwise.

Function: *image_mod_copy*()

Subserver: modhist

Syntax:

```
int image_con_copy(fd , ibuf , obuf);
int fd , ibuf , obuf;
```

Description:

Copies buffer *ibuf* into buffer *obuf* in the modhist subserver.

Return Value:

0 on error.

1 otherwise.

Function: *image_avg*()

Subserver: con

Syntax:

int image_avg(fd , ibuf , obuf);

int fd , ibuf, obuf;

Description:

Averages the image in buffer *ibuf* and the output is stored in buffer *obuf* of the con subserver.

Return Value:

0 on error.

1 otherwise.

Function: *image_con3*()

Subserver: con

Syntax:

int image_con3(fd , ibuf , obuf , mask);

int fd, ibuf, obuf;

IFMASK mask;

Description:

Covolves the image in buffer *ibuf* of the con subserver with mask and the output image is stored in buffer *obuf*.

Return Value:

0 on error.

1 otherwise.

Function: *image_horfil*() , *image_verfil*(), *image_diafil1*(), *image_diafil2*()

Subserver: con

Syntax:

```
int image_horfil(fd, ibuf, obuf);
int image_verfil(fd, ibuf, obuf);
int image_diafil1(fd, ibuf, obuf);
int image_diafil2(fd, ibuf, obuf);
int fd, ibuf, obuf;
```

Description:

Carries out horizontal, vertical, diagonal filtering respectively on image stored in buffer *ibuf* of the con subserver. image_diafil1() filters along the principal diagonal whereas, image_diafil2() filters along the other diagonal.

Return Value:

0 on error.

1 otherwise.

Function: *image_lp1()*, *image_lp2()*, *image_lp3()*

Subserver: con

Syntax:

```
int image_lp1(fd, ibuf, obuf);
int image_lp2(fd, ibuf, obuf);
int image_lp2(fd, ibuf, obuf);

int fd, ibuf, obuf;
```

Description:

These functions apply various laplacian masks to image in buffer *ibuf* of the con subserver to perform edge detection. The output is stored in buffer *obuf*.

Return Value:

0 on error.

1 otherwise.

Function: *image_median()*

Subserver: con

Syntax:

int image_median(fd, ibuf, obuf);

int fd, ibuf, ouf;

Description:

This function applies median filetring on the image stored in buffer *ibuf* and stores the output image into the buffer obuf of con subserver.

Return Value:

0 on error.

1 otherwise.

Function: *image_sobel()*

Subserver: con

Syntax:

int image_sobel(fd, ibuf, obuf);

int fd, ibuf, obuf;

Description:

Sobel edge detection operator is applied to the input image stored in buffer *ibuf* of the con subserver and the output image is stored in the buffer *obuf*.

Return Value:

0 on error.

1 otherwise.

Function: *image_prewitt()*

Subserver: con

Syntax:

int image_prewitt(fd, ibuf, obuf);

int fd, ibuf, obuf;

Description:

Prewitt nonlinear edge detection is performed on the image stored in buffer *ibuf* of the con subserver and the resultant image is kept in buffer *obuf*.

Return Value:

0 on error.

1 otherwise.

Function: *image_sh1()*, *image_sh290*, *image_sh3()*

Subserver: con

Syntax:

```
int image_sh1(fd, ibuf, obuf);
int image_sh2(fd, ibuf, obuf);
int image_sh3(fd, ibuf, obuf);

int fd, ibuf, obuf;
```

Description:

Various edge sharpening masks are applied to the input image stored in buffer *ibuf* of the con subserver and the sharped image is stored in the buffer *obuf*.

Return Value:

0 on error.

1 otherwise.

Function: *image_nc1()*, *image_nc2()*

Subserver: con

Syntax:

```
int image_nc1(fd, ibuf, obuf);
int image_nc2(fd, ibuf, obuf);

int fd, ibuf, obuf;
```

Description:

Noise cleaning masks are applied to the input image in buffer *ibuf* and the output image is kept in buffer *obuf* of the con subserver.

Return Value:

0 on error,

1 otherwise.

Function: *image_cg_n()*, *image_cg_e()*, *image_cg_w()*, *image_cg_s()*, *image_cg_ne()*, *image_cg_nw()*, *image_cg_sw()*, *image_cg_se()*

Subserver: con

Syntax:

```
int image_cg_n(fd, ibuf, obuf);
int image_cg_e(fd, ibuf, obuf);
int image_cg_w(fd, ibuf, obuf);
int image_cg_s(fd, ibuf, obuf);
int image_cg_ne(fd, ibuf, obuf);
int image_cg_nw(fd, ibuf, obuf);
int image_cg_sw(fd, ibuf, obuf);
int image_cg_se(fd, ibuf, obuf);

int fd, ibuf, obuf;
```

Description:

The compass gradient masks are applied to the input image. The input image and the output are stored in buffers *ibuf* and *obuf* respectively in the con subserver.

Return Value:

0 on error,

1 otherwise.

Function: *image_add()*

Subserver: pnt

Syntax:

```
int image_add(fd, ibuf1, ibuf2, obuf);
int fd, ibuf1, ibuf2, obuf;
```

Description:

Images stored in buffers *ibuf1* & *ibuf2* are added pixelwise and the output image is stored in buffer *obuf* of the pnt subserver. If the resultant gray value of any pixel exceeds maximum gray value, it is rounded off to the maximum.

Return Value:

0 on error,

1 otherwise.

Function: *image_sub()*

Subserver: pnt

Syntax:

int image_sub(fd, ibuf1, ibuf2, obuf);

int fd, ibuf1, ibuf2, obuf;

Description:

Image stored in buffer *ibuf2* is subtracted from buffer *ibuf1* pixelwise and the absolute differences form the output image which is stored in buffer *obuf* of the pnt subserver.

Return Value:

0 on error,

1 otherwise.

Function: *image_stretch()*

Subserver: pnt

Syntax:

int image_stretch(fd, ibuf, obuf, range);

int fd, ibuf, obuf;

IRANGE range;

Description:

The range of the gray values of input image which is stored in the buffer of the pnt subserver is stretched/compressed linearly. The IRANGE structure contains the range to stretch from and the range to stretch to. The output is stored in the output buffer *obuf* of the pnt subserver.

Return Value:

0 on error,

1 otherwise.

Function: *image_neg()*

Subserver: pnt

Syntax:

int image_neg(fd, ibuf, obuf);

int fd, ibuf, obuf;

Description:

Negative of the input image is found and stord in buffer *obuf* of the pnt

subserver. The input image is taken from buffer *ibuf*.

Return Value:

0 on error,

1 otherwise.

Function: *image_slice()*

Subserver: pnt

Syntax:

```
int image_slice(fd, ibuf, obuf, bitnum);
int fd, ibuf, obuf, bitnum;
```

Description:

Bit slicing is performed on the input image in buffer *ibuf* of the pnt subserver. bitnum is the bit number to be sliced, which lies between 0-7. The output image is stored in buffer *obuf* of the same subserver.

Return Value:

0 on error,

1 otherwise.

Function: *image_norm()*

Subserver: modhist

Syntax:

```
int image_norm(fd, ibuf, obuf, mean, sdev);
int fd, ibuf, obuf, mean, sdev;
```

Description:

Histogram of the input image is modified so that the resultant histogram is closest possible to the gaussian function with mean *mean* and standard deviation *sdev*. Input image is taken from buffer *ibuf* of modhist subserver and output is kept in buffer *obuf* of the same.

Return Value:

0 on error,

1 otherwise.

Function: *image_uni()*

Subserver: modhist

Syntax:

```
int image_uni(fd, ibuf, obuf);
int fd, ibuf, obuf;
```

Description:

Performs histogram equalization on the input image in buffer *ibuf* of the modhist server and the equalized image stored in the buffer *obuf* of the same subserver.

Return Value:

0 on error,

1 otherwise.

Function: *image_exp()*

Subserver: modhist

Syntax:

```
int image_exp(fd, ibuf, obuf, alpha);
int fd, ibuf, obuf;
float alpha;
```

Description:

The input image is modified so that the histogram of the resultant image closely resembles the exponential distribution with parameter alpha. The input image is taken from buffer *ibuf* and output image is stored in the buffer *obuf* of the modhist server.

Return Value:

0 on error,

1 otherwise.

Function: *image_hyp_cbrt()*, *image_hyp_log()*

Subserver: modhist

Syntax:

```
int image_hyp_cbrt(fd, ibuf, obuf);
int image_hyp_log(fd, ibuf, obuf);

int fd, ibuf, obuf;
```

Description:

Histogram hyperbolization with cuberoot/ logarithmic density functions is performed on the image in buffer *ibuf* of the modhist subserver and the modified image is stored in buffer *obuf* of the smae subserver.

Return Value:

0 on error,

1 otherwise.

Function: *image_rayleigh()*

Subserver: modhist

Syntax:

int image_rayleigh(fd, ibuf, obuf, alpha);

int fd, ibuf, obuf;

float alpha;

Description:

The image is modified so that the histogram of the output image ic closest possible to rayleigh density function with parameter alpha. The input image is taken from buffer *ibuf* and the output image is stored in buffer *obuf* of the modhist subserver.

Return Value:

0 on error,

1 otherwise.

Appendix E

Performance Analysis

The performance of the image processing library, *ilib*, was measured using the time functions offered by Unix. *ilib* notes the time before distributing tasks over the network and again after collection of individual results. No claim is made about these timings for being invariant under different load conditions of the network. But they represent a trend which is invariant of the load.

Three graphs (E.1 to E.3) are attached in the following pages which give the comparison between times required for a single processor and 2, 4, 8 processors respectively. The data points fluctuate about the line of best fit, the slope of which gives the average speedup obtained. The y-intercept of this line denotes the average setting up time for an image processing operation. This includes the communication overhead for distributing the job and time for dividing the image. For operations which involve less computation, it is seen that the distribution load plays an important role and the speedup obtained is far below linear. Speedup is defined as the ratio of time taken by the task for multiple processors and that by a single processor.

Graphs E.4 to E.6 show the speedup for 2, 4, 8 processors respectively. On the X-axis is the operation number which is plotted against the speedup obtained for that operation with different number of processors. It can be easily seen from these graphs that the optimal number of processors required for maximum speedup lies between 4 and 8.

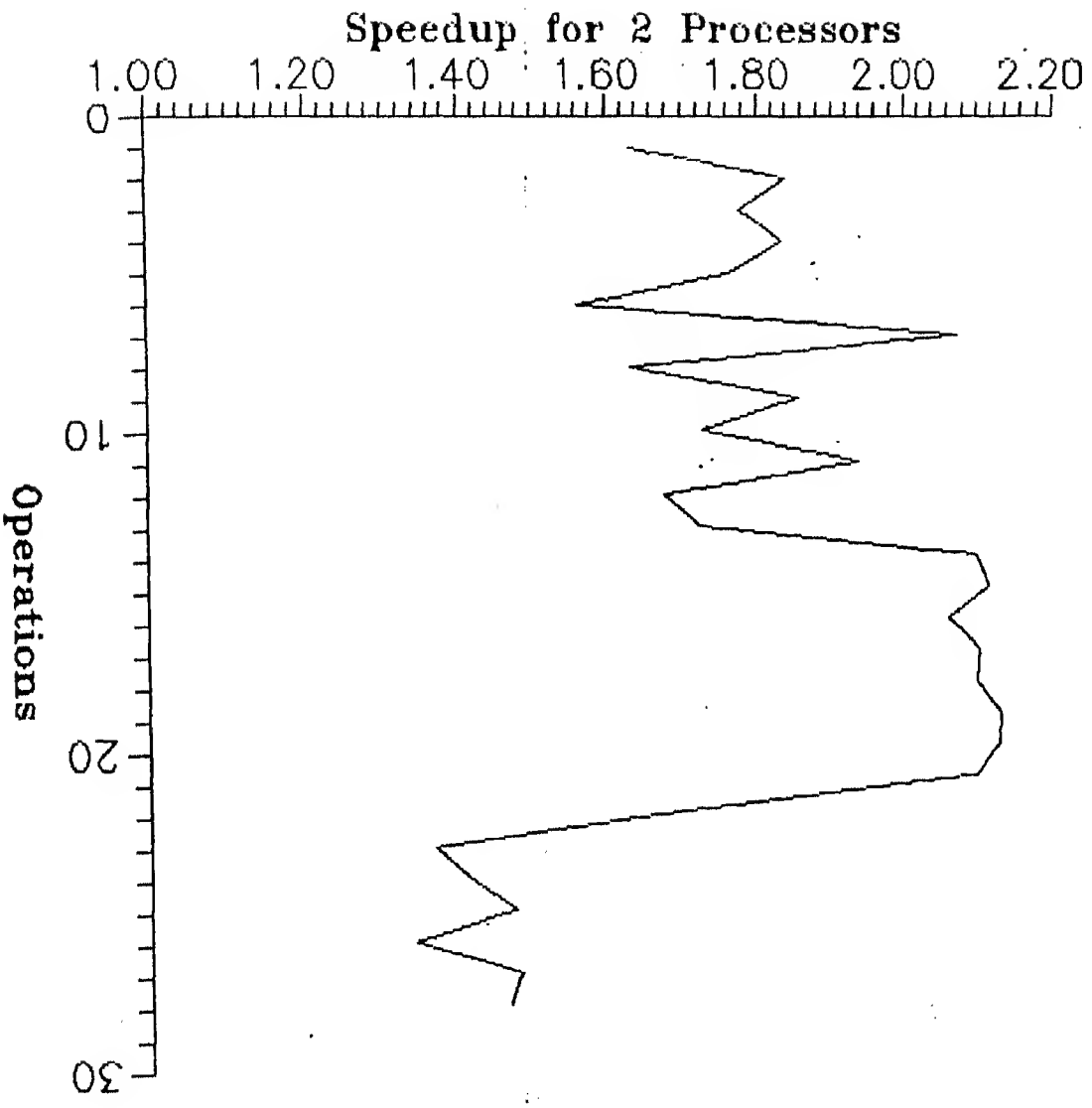
Operations specifically dealing with files show negative trends in speedup as the number of processor grow. This is because of the exclusive access to the file while writing and reading. Another cause for this is the statelessness of the NFS server. If the files were distributed over diffrent NFS servers, parallelism could be exploited in these operations too.

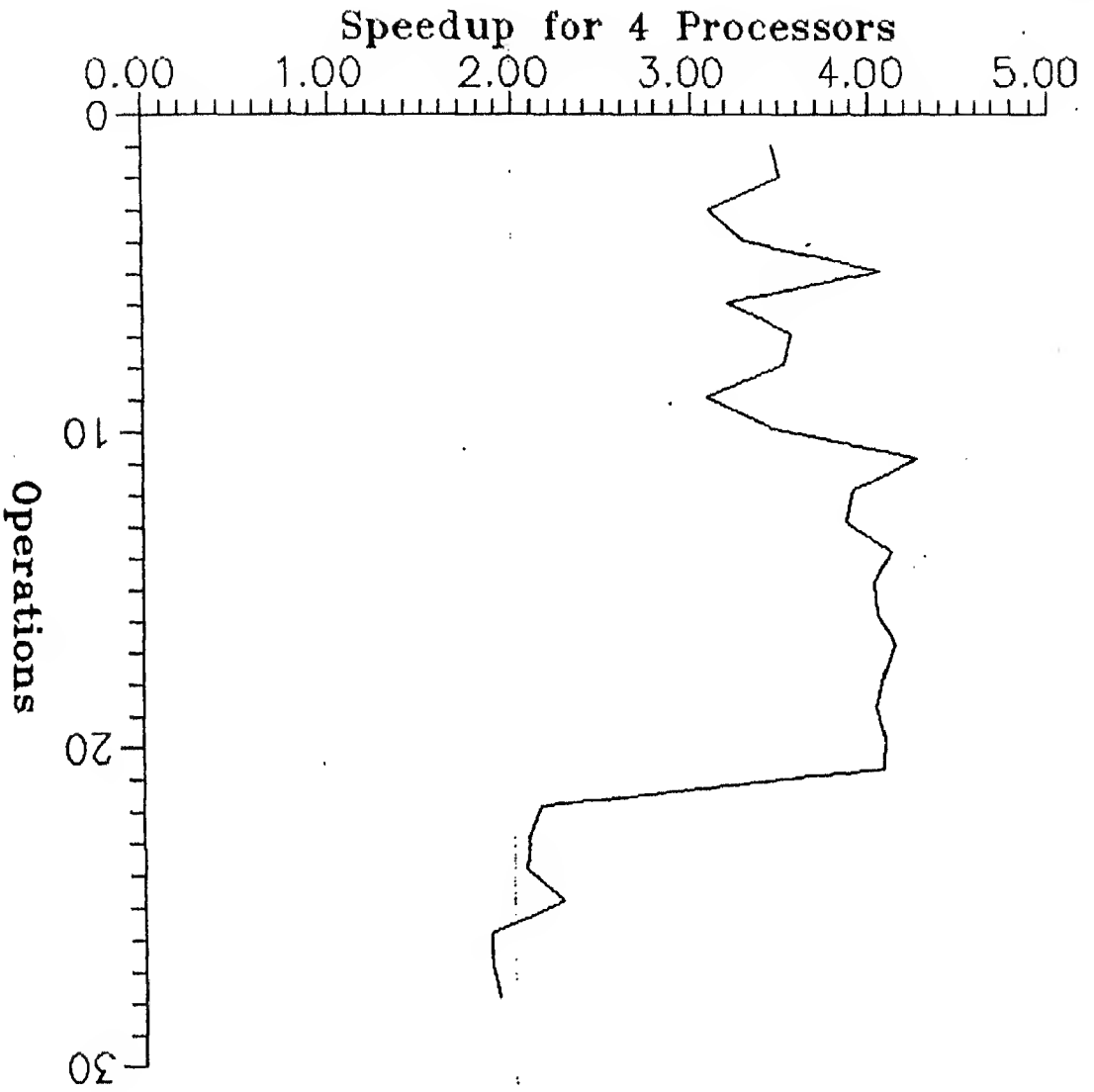
E.1

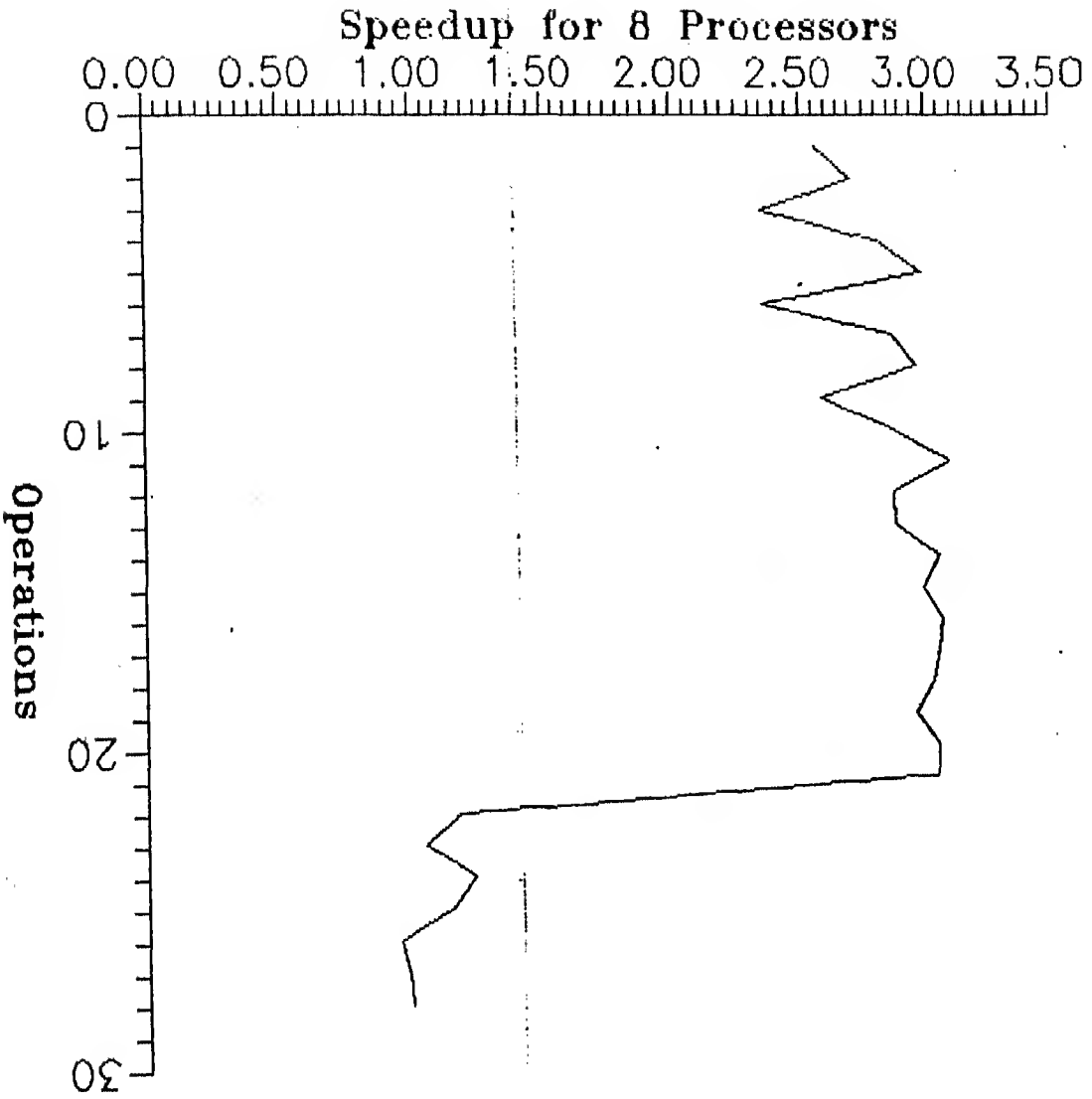
Time for 2 Processors (ms)
0.00
5000.00
10000.00
15000.00
20000.00
Time for 1 Processor (ms)
0.00
5000.00
10000.00
15000.00
20000.00
25000.00
30000.00
35000.00

Time for 4 Processors (ms)
0.00
2000.00
4000.00
6000.00
8000.00
10000.00
Time for 1 Processor (ms)
0.00
5000.00
10000.00
15000.00
20000.00
25000.00
30000.00
35000.00

Time for 8 Processors (ms)
0.00
5000.00
10000.00
15000.00
20000.00
Time for 1 Processor (ms)
0.00
5000.00
10000.00
15000.00
20000.00
25000.00
30000.00
35000.00


Speedup for 2 Processors
1.00
1.20
1.40
1.60
1.80
2.00
2.20
0
10
20
30
Operations


Speedup for 4 Processors
0.00
1.00
2.00
3.00
4.00
5.00
0
10
20
30
Operations


Speedup for 8 Processors
0.00 0.50 1.00 1.50 2.00 2.50 3.00 3.50
0
10
20
30
Operations